UTS UNIVERSITY OF TECHNOLOGY SYDNEY

Efficient and Reproducible Automated Deep Learning

by Xuanyi Dong

Thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

under the supervision of
Prof. Bogdan Gabrys and Prof. Katarzyna Musial

University of Technology Sydney
Faculty of Engineering and Information Technology

Apr 2021

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, **Xuanyi Dong** declare that this thesis, is submitted in fulfilment of the requirements for the award of **Doctor of Philosophy**, in the **School of Computer Science, FEIT** at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: **28 Apr 2021**

ABSTRACT

Efficient and Reproducible Automated Deep Learning

by

Xuanyi Dong

Deep learning has shown its power in a large number of applications, such as visual perception, language modeling, speech recognition, video games, etc. To deploy a deep learning model successfully, inevitable manual tuning is required for each component, such as neural architecture design, the choice of optimization strategy, data selection, augmentation, etc. Such manual tuning costs expensive computational resources and is labor-intensive. Moreover, this paradigm is not scalable when the model size or the data size significantly increases. Fortunately, AutoDL brings hope to alleviate this problem by making the tuning procedure automated. Despite the recent success of AutoDL, efficiency and reproducibility for AutoDL algorithms remain a tremendous challenge for the community.

In this thesis, we address this challenge in the following aspects. We comprehensively review the current state of AutoDL and set up six step-by-step objectives to further develop AutoDL. To achieve these objectives, we propose a series of efficient approaches to learning to search (1) neural architecture topology, (2) neural architecture size, and (3) hyperparameters by gradient descent. In addition to common empirical analysis on vision and NLP datasets, we build a systematical benchmark for neural architecture topology and neural architecture size. This benchmark aims to provide a fair and easy-to-use environment for our proposed algorithms as well as other AutoDL participants.

Dissertation directed by Professor Bogdan Gabrys
Advanced Analytics Institute, Faculty of Engineering and IT, University of Technology Sydney

Acknowledgements

First and foremost, I would like to thank my academic supervisor Prof. Bogdan Gabrys. I am incredibly grateful for his kind, timely, and strong support. He guided me on automated deep learning, which is my favorite research direction; he let me know what good research work is; he also kindly shared many useful suggestions regarding career development. Since 2015, when I first got in touch with deep learning, I was lucky to be mentored by many outstanding researchers. Thanks to Dr. Junjie Yan, my mentor, when I interned at SenseTime, from whom I learned how to get a good ranking at large-scale research competitions. Thanks to Prof. Shuicheng Yan and Dr. Junshi Huang, my mentors, when I interned at Qihoo 360, where I published my first paper. Thanks to Prof. Deyu Meng, my mentor, when I visited at Xi'an Jiaotong University, from whom I learned the rigorous attitude and mathematical expression of doing research. Thanks to Prof. Yaser Sheikh and Dr. Shoou-I Yu, my mentors, when I interned at Facebook Reality Labs, from whom I learned how to perform research in a systematic way. Thanks to Dr. Yan Yan, my senior, from whom I learned many machine learning methodologies and shared many experiences when I was a junior. Thanks to Prof. Yi Yang for the scholarship support of one and a half year's study at the University of Technology Sydney and his recommendation for the internship and fellowship. Thanks to Dr. Quoc V. Le and Dr. Mingxing Tan, my hosts, when I interned at Google Brain, from whom I learned critical thinking and what is the research impact. Thanks to Mr. Daiyi Peng, my co-host during my Google internship, from whom I systematically learned the industrial system and rethought the relationship between engineer and research. Thanks to Prof. Katarzyna Musial, my co-supervisor in UTS, who provided useful suggestions for my research on AutoDL.

I would also like to thank my colleagues and friends at the University of Technol-

ogy Sydney and other institutes. I want to thank Xiaojun Chang, Hehe Fan, Qianyu Feng, Qingji Guan, Yang He, Yanbin Liu, Ping Liu, Yutian Lin, Peike Li, Fan Ma, Guang Li, Guangrui Li, Jiaxu Miao, Pingbo Pan, Ruijie Quan, Tianqi Tang, Bingwen Hu, Minfeng Zhu, Yu Wu, Xiaohan Wang, Zhongwen Xu, Guoliang Kang, Yan Yan, Zongxin Yang, Fengda Zhu, Hu Zhang, Zhun Zhong, Zhedong Zheng, Liang Zheng, Xiaolin Zhang, Tianjian Meng, Adams Wei Yu, Arber Zela, Cihang Xie, Xinshuo Weng, and many others. I was very fortunate to collaborate with or discuss with them. These discussions inspired and motivated many of my research works.

I would also like to thank Data to Decision CRC, CAI FEIT UTS, AAI FEIT UTS, Baidu Scholarship, and Google PhD Fellowship for supporting my research.

Lastly, I would like to thank my father, Aimin Dong, and my wife Lu Liu for their support and love throughout the years. Especially without taking any credits, Lu helped me a lot in discussing new research ideas, designing experiments, and re-writing the papers.

Xuanyi Dong

Sydney, Australia.

List of Publications

Selected Journal Papers

J-[1] **Xuanyi Dong**, Lu Liu, Katarzyna Musial, Bogdan Gabrys. "NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size", *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (ERA CORE Rank A*, IF=17.861)

J-[2] **Xuanyi Dong**, Yi Yang, Shih-En Wei, Xinshuo Weng, Yaser Sheikh, Shoou-I Yu. "Supervision by Registration and Triangulation for Landmark Detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (ERA CORE Rank A*, IF=17.861)

J-[3] **Xuanyi Dong**, Liang Zheng, Fan Ma, Yi Yang, Deyu Meng. "Few-Example Object Detection with Model Communication", *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (ERA CORE Rank A*, IF=17.861)

J-[4] **Xuanyi Dong**, Yan Yan, Mingkui Tan, Yi Yang, Ivor W. Tsang. "Late Fusion via Subspace Search with Consistency Preservation", *IEEE Transactions on Image Processing (TIP)* (ERA CORE Rank A*, IF=9.34)

Selected Conference Papers

C-[5] [NAS@ICLR-2021] **Xuanyi Dong**, Mingxing Tan, Adams Wei Yu, Daiyi Peng, Bogdan Gabrys, Quoc V. Le. "AutoHAS: Efficient Hyperparameter and Architecture Search", Workshop on Neural Architecture Search (NAS) at International Conference on Learning Representations (ICLR)

C-[6] [ICLR-2020] **Xuanyi Dong**, Yi Yang. "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search", International Conference on Learning Representations (ICLR) (H-index=150)

C-[7] [NeurIPS-2020] Daiyi Peng, **Xuanyi Dong**, Esteban Real, Mingxing Tan, Yifeng Lu, Hanxiao Liu, Gabriel Bender, Adam Kraft, Chen Liang, Quoc V. Le. "PyGlove: Symbolic Programming for Automated Machine Learning", *Advances in Neural Information Processing Systems (NeurIPS)* (ERA CORE Rank A*)

C-[8] [NeurIPS-2019] **Xuanyi Dong**, Yi Yang. "Network Pruning via Transformable Architecture Search", *Advances in Neural Information Processing Systems (NeurIPS)* (ERA CORE Rank A*)

C-[9] [ICCV-2019] **Xuanyi Dong**, Yi Yang. "One-Shot Neural Architecture Search via Self-Evaluated Template Network", *IEEE Conference on International Conference on Computer Vision (ICCV)* (ERA CORE Rank A*)

C-[10] [ICCV-2019] **Xuanyi Dong**, Yi Yang. "Teacher Supervises Students How to Learn from Partially Labeled Images for Facial Landmark Detection", *IEEE Conference on International Conference on Computer Vision (ICCV)* (ERA CORE Rank A*)

C-[11] [CVPR-2019] **Xuanyi Dong**, Yi Yang. "Searching for A Robust Neural Architecture in Four GPU Hours", *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (ERA CORE Rank A*)

C-[12] [CVPR-2018] **Xuanyi Dong**, Shoou-I Yu, Xinshuo Weng, Shih-En Wei, Yi Yang, Yaser Sheikh. "Supervision-by-Registration: An Unsupervised Approach to Improve the Precision of Facial Landmark Detectors", *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (ERA CORE Rank A*)

C-[13] [CVPR-2018] **Xuanyi Dong**, Yan Yan, Wanli Ouyang, Yi Yang. "Style Aggregated Network for Facial Landmark Detection", *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (ERA CORE Rank A*)

C-[14] [CVPR-2017] **Xuanyi Dong**, Junshi Huang, Yi Yang, Shuicheng Yan. "More is Less: A More Complicated Network with Less Inference Complexity", *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (ERA CORE

Rank A*)

Contents

List of Figures

List of Tables

Abbreviation

ML: Machine Learning

DL: Deep Learning

NN: Neural Network

CNN: Convolutional Neural Network

RNN: Recurrent Neural Network

NAS: Neural Architecture Search

HPO: Hyperparameter Optimization

AutoML: Automated Machine Learning

AutoDL: Automated Deep Learning

RL: Reinforcement Learning

ES: Evolutionary Strategy

FLOP: FLoating Point Operation

GPU: Graphics Processing Unit

LSTM: Long Short-Term Memory

AOS: Alternative Optimization Strategy

SVD: Singular Value Decomposition

PPO: Proximal Policy Optimization

Chapter 1

Introduction

Since the birth of AlexNet [16] in 2012, deep learning technologies [17] have advanced the state-of-the-art performance of copious applications, such as object detection [18, 3], landmark localization [13, 10, 12], machine translation [19], classification [20, 4] etc. To apply deep learning technologies to an application successfully, substantial human effort is required in the pipeline of data collection and cleaning, model and algorithm design, hyperparameter choices, optimization, deployment. With the development in the last decade, these steps have involved much human prior knowledge and became more and more complex. Moreover, with the increased complexity, it is more and more difficult for a human expert to continue improving existing methods; thus the headroom for improvements of manual tuning is saturated. For example, the accuracy gain of the best image classification model from the 2014 year to the 2015 year is about 10% on ImageNet [20], whereas the gain from the 2016 year to the 2017 year is just about 2%. Such slowing down is because the manual tuning is unfeasible for almost infinite possibilities.

Fortunately, automated deep learning – AutoDL – brings the hope to address the aforementioned problem. Formally speaking, AutoDL is intended to automate the process of applying deep learning to real-world problems, which included but is not limited to model selection [21], neural architecture search [22], hyperparameter optimization [23]. A typical and general AutoDL framework consists of three steps. Firstly, defining the search space for the process to be automated, e.g., all possible candidate architectures if automating neural architecture design. Secondly,

sampling one or multiple candidates over this search space. Thirdly, evaluating the sampled candidates to get their performance and using the performance to guide the next sampling. The random search method [24], Bayesian optimization [25], EA method [26], RL method [27], efficient NAS with parameter sharing [28, 29, 11], and others for AutoDL can be viewed as a special case of this framework.

The study of such a framework has a long history, dating back to the 1990s [30, 31], which usually refers to automated machine learning – AutoML. When the deep learning era came, to accommodate the characteristics of deep learning, a new and special direction within AutoML – AutoDL – was developed. Most AutoDL algorithms are a direct extension of AutoML algorithms with additional modifications for deep learning. Since a deep learning algorithm usually requires much more computational resources than the traditional machine learning algorithm, the cost of a single trial (evaluating a candidate and obtaining the performance) is significantly increased; not to mention the massive cost for thousands of trials required by an AutoDL algorithm. As a result, most of these AutoDL algorithms have not been accessible to a broader AutoDL community. For example, it takes over 2000 GPU days to run the PPO algorithm [32] to automatically discover a high-accuracy CNN on a small vision dataset [33].

A natural question is: *could we reduce the massive computational cost of AutoDL algorithms – efficient AutoDL?* Let us take a closer look at the AutoDL framework. Its cost roughly equals the multiplication of a single evaluation's cost and the number of conducted evaluations. To reduce the overall cost of the AutoDL algorithm, we must reduce the cost of obtaining the performance of a single deep learning algorithm and also reduce the number of evaluations required by an AutoDL algorithm. The key for the former is accurately approximating a deep learning algorithm's performance with an affordable cost. The key for the latter is intelligently selecting useful trials and avoiding low-performance trials. We will discuss these in more

details in Chapter 3.

Another challenge to enable efficient AutoDL is its generalization ability. AutoDL is a broad concept that includes neural architecture search, hyperparameter optimization, etc. Efficient AutoDL requires to incorporate prior knowledge to automate the design of a specific deep learning component. As such prior knowledge for one component (e.g., architecture topology) may differ from that for another (e.g., hyperparameters), an efficient AutoDL algorithm for architecture is not applicable to hyperparameters or data collection. In other words, an efficient AutoDL algorithm is usually customized for a single component and lacks generalization ability for other components. However, it is impractical to design one efficient AutoDL algorithm for each of the tens of components in a deep learning pipeline.

A new research question arises: *could we design a unified and efficient AutoDL framework* that is applicable to each component in DL, e.g., architecture and hyperparameters? Such a framework should be able to handle a broader and more practical search space by combining the search spaces of each individual component. The architecture choices in a DL pipeline are often categorical, whereas the hyper-parameter choices are often continuous; others might be even more complex. Therefore, a critical challenge here is how to handle these different types of values in a joint search space. There are also other challenges to design a unified and efficient AutoDL framework and they will be discussed in Chapter 4.

Apart from the perspective of the AutoDL framework/algorithm, many researchers worried about the reproducibility of AutoDL research [34]. Please recall that AutoDL has three important elements: search space, search algorithm, and a child deep learning program (a single trial). There are many technical details in each element, such as how to create the search space, the meta hyperparameter in the search algorithm, and the re-training strategy in the child deep learning program.

Given the limited space in a typical research paper submission, it is hard to expose all of these details, yet some of them are critical to the final performance of an AutoDL algorithm[*]. Having said that, the missing details often make the reproducability of the results of an AutoDL algorithm impossible. Moreover, a variety of algorithms search for neural architectures using different search spaces. These searched architectures are trained using different setups, such as hyperparameters, data augmentation, regularization. This also raises a comparability problem when comparing the performance of various search algorithms.

The third research question can therefore be formulated as: *how could we enable reproducible AutoDL?* A straightforward solution is to force researchers to report as many technical details as possible, especially the critical ones. However, it is hard for authors to report all the critical details and also difficult for reviewers to find missing details. To solve this problem, a simple research protocol is desired for AutoDL researchers. Here, we suggest two directions. The first is to build the off-the-shelf dataset of deep learning programs and use it to validate the AutoDL algorithm. For example, if we build an architecture dataset with the off-the-shelf accuracy information on each dataset, then neural architecture search methods can directly query the performance of an architecture candidate from this dataset. In this way, the effect caused by different re-training strategies can be avoided. The second is to build a unified and modularized codebase for AutoDL. This codebase should provide popular state-of-the-art AutoDL solutions and be easy to extend for future research. The modification/extension of a new algorithm based on this codebase could be decoupled, and thus the contribution of each modification to the final performance can be fairly evaluated. With the help of such datasets and

[*]The performance of an AutoDL algorithm includes but is not limited to the convergence speed, the resource requirement, the accuracy of the final discovered deep learning program, the FLOPs/latency of the final discovered deep learning program, etc.

codebase, the reproducible problem could be addressed. We will discuss this in more details in Chapter 5.

In conclusion, this thesis project aims to answer these research questions:

Q-1 *How could we reduce the massive computational cost of AutoDL algorithms?*

Q-2 *How could we design a unified and efficient AutoDL framework?*

Q-3 *How could we enable reproducible AutoDL?*

To answer them, we identified the following six research objectives and introduced how they are associated with the above questions.

Obj-1 *Critically review the AutoDL literature and analyze the AutoDL algorithms.* This could let us understand the advantages and disadvantages of existing solutions; therefore, this objective is a prerequisite to solve Q-1, Q-2, and Q-3.

Obj-2 *Design efficient AutoDL algorithm for searching for neural architecture topology.* This objective could answer Q-1 from the methodology perspective.

Obj-3 *Systemically evaluate the proposed efficient algorithm on computer vision and natural language processing applications.* This objective could empirically demonstrate the effectiveness of the approach proposed in Obj-2 on a wide range of applications. If the empirical results show its superiority, it means "yes" to Q-1.

Obj-4 *Generalize the proposed efficient algorithm to handle both neural architecture topology and neural architecture size.* This objective aims to design a coherent approach based on Obj-2. Ideally, the new approach could be used to discover every aspect of neural architecture. If so, it means we could extend the scope of neural topology search to a broader scope of neural architecture search. This is the first step towards unified AutoDL.

Obj-5 *Generalize the proposed efficient algorithm from the search for the architecture only to also include the hyperparameters.* This objective is more ambitious compared to Obj-4. If we can propose such a joint search algorithm, it is very strong evidence for the existence of an efficient and unified AutoDL framework.

Obj-6 *Build large-scale architecture datasets to encourage reproducible neural architecture search in the AutoDL community.* In parallel to Obj-1 to Obj-5, this objective aims to enhance the foundation of AutoDL, because a fair and easy-to-use benchmark can facilitate people to do scientific research.

To accommodate with these objectives, this thesis coherently describes and comprehensively analyzes the proposed methodologies through Chapter 2 to Chapter 5, and made a conclusion in Chapter 6.

The overall methodological approach and steps undertaken in the project are presented below. In Chapter 2, we achieved the Obj-1. Specifically, the history of AutoDL is reviewed and the advantages and disadvantages of different AutoDL algorithms and systems are discussed. A part of the discussed material and methods in this Chapter have been published in peer-reviewed conferences (C-[11], C-[6], C-[7]).

In Chapter 3, we achieved the Obj-2, Obj-3, and Obj-4. Specifically, efficient AutoDL algorithms to search for neural architecture topology and neural architecture size are proposed. The original contributions include the proposed novel gradient-based architecture sampler, the fastest neural architecture search algorithm in the year 2018, a new NAS-based paradigm for network pruning, and state-of-the-art performance for network pruning. These contributions have been accepted and published in the peer-reviewed conferences CVPR and NeurIPS (C-[11] and C-[8]).

In Chapter 4, we achieved the Obj-5. Specifically, the efficient AutoDL algorithm described in Chapter 3 has been generalized to handle both architecture and hyper-

parameters. The original contributions include the proposed unified and efficient AutoDL framework and the systemic, empirical analysis of seven different deep learning models on large-scale datasets. This work has been accepted to NAS@ICLR 2021 [5].

In Chapter 5, we achieved the Obj-6. Specifically, an algorithm-agnostic NAS benchmark has been built with information of 15K neural cell candidates for architecture topology and 32K for architecture size on three vision datasets. In addition, a fair evaluation of implemented in a single codebase 13 state-of-the-art NAS approaches have been conducted and can be used as baselines. Both the results and codebase are publicly available to the community. The preliminary version of this benchmark has been published at ICLR 2020 (C-[6]) and the full version was accepted to IEEE TPAMI (J-[1]).

In Chapter 6, the thesis conclusion and discussion of the future research directions for AutoDL are provided.

Chapter 2

AutoDL: A Critical Review

Machine learning is the study of computer algorithms that improve automatically through experience [35] and has a long history that can be traced back to the 1950s. Until the year 2012, traditional approaches (such as logistic regression, decision tree, K-means, support vector machine) dominated machine learning. Later, the deep learning approach showed superior performance in a number of important areas and gradually became mainstream.

Generally speaking, machine learning, especially deep learning, algorithms can be used in a wide variety of applications, such as computer vision [36, 37, 3, 18], natural language processing [38], recommendation [39], etc. In Figure 2.1, a life cycle of a deep learning algorithm is illustrated. Typically, it consists of four steps: (1) *human* collect raw data; (2) *human* pre-process the collected data, such as clean, normalization, augmentation, etc; (3) *human* select deep learning model and optimization strategy to optimize it; (4) *human* deploy the optimized model to production environment (e.g., iPhone) that is designed by *human*; (optionally 5) the deployed model might need to be online adapted according to streaming data. The human expert is heavily involved in this life cycle.

Back to 2012, with the rebirth of deep learning, there are numerous opportunities for researchers to improve each part of the deep learning life cycle. For example, residual connection improves the ImageNet classification accuracy by 7% [20]; transformer structure improves the BLEU score by 2 [19]; warm restart schedule consistently works well for hundreds of tasks [40]. However, in 2021, low-hanging

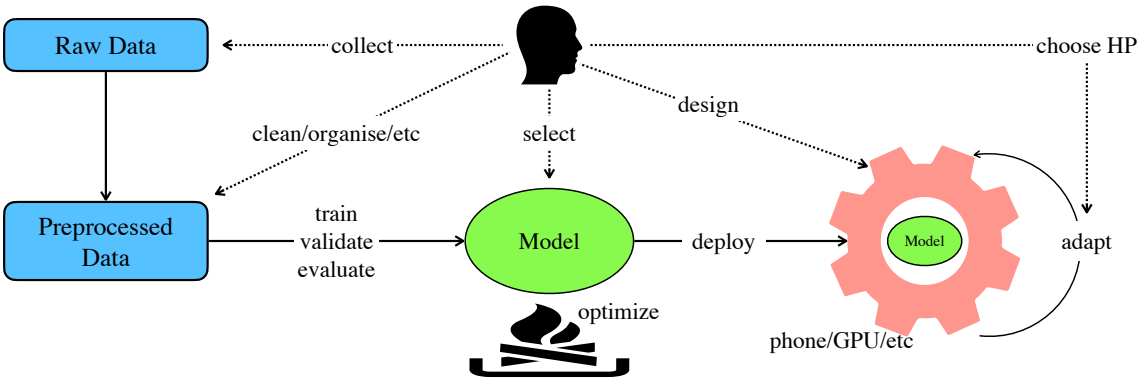


Figure 2.1 : The life cycle of a deep learning algorithm.

fruits in deep learning are picked, and each part in the deep learning life cycle is heavily optimized by human experts. Consequently, the headroom of performance gain for a deep learning algorithm is limited, if we still rely on human experts to manually select models, choose hyperparameters, design chips, etc.

Such a limitation urge deep learning players to revolutionize how human interacts with the deep learning life cycle. AutoDL is the most promising direction to solve this problem. AutoDL aims to automate the manual design for each part of the deep learning life cycle. The terminology "AutoDL" appears in the year 2018, while its study has a long history. AutoDL – as a special case of AutoML – can utilize all AutoML algorithms, thus can trace back to the 1990s [30]. Having said that, most AutoML works studied on traditional algorithms, such as linear regression or support vector machine [31]. AutoDL is more challenging due to the much complicated deep learning model and the expensive cost of training a single deep learning model. Customized AutoML algorithms for deep learning model started at around the year 2011 [41, 42]. However, until the year 2015, most of these approaches play with small-scale neural networks [43, 44]. In 2016, finally, researchers successfully showed the promising results of AutoML on deep neural networks [45, 33][*].

[*]Although these works are published at the 2017 venues, they were online available in 2016.

The ultimate goal of AutoDL is to automate everything with minimal human interaction. The AutoDL Challenge [46] made a substantial effort in this direction – organizing a competition with around 100 vision, NLP, speech datasets. AutoML-Zero [47] studied how to find everything from scratch without human experts. Since such goal is ambitious and far from practice, most AutoDL researches focus on automating a small piece in the deep learning life cycle, such as neural architecture search [28, 33, 29, 11], hyperparameter optimization [43, 48, 49, 5], hardware search [50], etc.

The rest of this chapter will discuss these AutoDL works categorized by their automation target. Specifically, the AutoDL algorithms for neural architecture design will be discussed in Section 2.1, that for hyperparameter design will be discussed in Section 2.2, the AutoDL benchmark will be discussed in Section 2.3, and the software designed for AutoDL will be discussed in Section 2.4.

2.1 Neural Architecture Search (NAS)

At the time of Jan 2021, NAS has become a dominant direction in AutoDL. Since this terminology "NAS" appeared at 2016, researchers have made significant progress in automatically discovering good architectures [33, 54, 29, 69, 70, 53, 11, 1]. Overall speaking, most NAS approaches could be summarized as the interaction between architecture search space, child deep learning program, and search algorithm as shown in Figure 2.2. We summarized the difference between each NAS algorithm in Table 2.1, and will separately discuss the development of these three components in below.

Child deep learning program is the basic in NAS, which controls how a neural architecture being trained and evaluated. In the early works [33, 45], researchers utilized the VGG-style [71] as the overall architecture structure and train each architecture with only a few epochs to approximate its performance. Later, apart

	Publicly Available	Search Space	Search Strategy	Improving Efficiency
[51]	2009.09	cell topology in LSTM	evolution	N/A
[52]	2015.07	topology and operation in LSTM	evolution	easy-to-hard tasks to filter
[33]	2016.11	filter size + connectivity	PPO	fewer epochs
[45]	2016.11	MetaQNN space	Q-learning	fewer epochs; early stop
[53]	2017.03	unrestricted CNN space	evolution	weight inheritance
[54]	2017.07	NASNet space	PPO	fewer epochs
[55]	2017.08	SMASH space	random	weight generation via HyperNet [56]
[57]	2017.10	activation functions	PPO	smaller model
[29]	2018.02	reduced NASNet space	REINFORCE	weight sharing [29]
[28]	2018.06	reduced NASNet space	differential	weight sharing; smaller model
[58]	2018.06	tree-structure	REINFORCE	Net2Net [59] technique
[60]	2018.07	MBConv-based space	PPO	fewer epochs
[61]	2018.07	modified NASNet space	random	weight sharing
[62]	2019.02	reduced NASNet space	random	weight sharing
[63]	2019.04	architecture generator space	manual	fewer epochs
[64]	2019.04	FPN space	PPO	smaller model; fewer epochs
[8]	2019.05	depth + width	differential	weight sharing
[11]	2019.06	reduced NASNet space	differential	weight sharing; Gumbel [65]
[66]	2019.12	scale-permuted space	PPO	smaller model; fewer epochs
[67]	2020.04	architecture generator space	Bayesian	fewer epochs
[68]	2020.04	normalization+activation	evolution	smaller dataset; fewer epochs
[5]	2020.06	MBS + hyperparameters (HP)	REINFORCE	weight sharing

Table 2.1 : We dissect different NAS algorithms from three characteristics.

from this direction, researchers also proposed to re-use the manually designed networks, such as MobileNet-V2 [15], and tune its configurations [60, 72]. The training hyperparameters may also be slightly different in different works [60, 54, 28, 11].

Search Space is an important research problem. The first several works [33, 45] made the connectivity, kernel size, number of channels, and others tunable. However, since such search space is too large to explore, the accuracy of the final discovered architecture is limited. Later on, researchers manually design the macro structure of a convolutional neural network and search for its micro (cell) structure [54, 60].

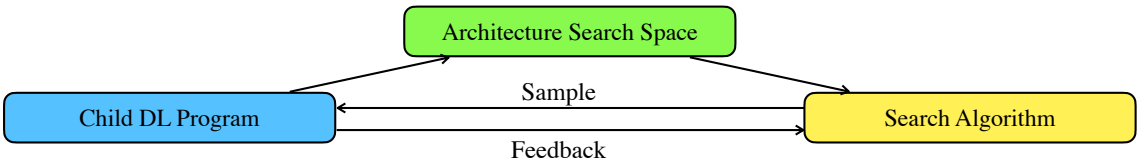


Figure 2.2 : We abstract the NAS approaches as an interaction between the child deep learning program, architecture search space, and search algorithm.

In this way, human expert knowledge is incorporated and the size of search space is constrained, and thus the NAS performance is significantly boosted. Two representative search spaces are "NASNet search space" and "MNasNet search space". Following that, some researchers simplifying them by removing the redundant architectures [73, 28, 72]. The aforementioned works focus on designing the convolutional neural network for the image classification task. There are also other works that explored how to design the search space for recurrent network [33, 28], transformer [74], segmentation [37], etc. More recent NAS works take one step ahead – studying how to automatically design search space [63, 75].

Search algorithms are responsible for exploring the search space. Many researchers utilized RL algorithms to sample candidate architectures from the search space [33, 54, 29, 60], where the RL agent is usually an RNN model that sequentially generate the architecture configuration. Another line of research employed the EA algorithms [53, 76, 74]. Other researchers utilized the Bayesian optimization [77, 78] or random search [79, 9, 80]. These works require to sample one (or multiple) candidate architectures – that treat each candidate as a discrete value. A novel research direction that relaxed such discrete search space and utilized the continuous relaxation of the architecture representation [28]. Therefore, these works [28, 11, 81, 8] can efficiently search architectures using gradient descent.

Efficiency. Since NAS algorithms usually require expensive computational re-

sources [33, 54, 76], an increasing number of researchers focus on improving the architecture search speed [82, 70, 29, 28]. A variety of techniques have been proposed, such as progressive-complexity search stages [70], accuracy prediction [83], HyperNet [55], Net2Net transformation [82], and parameter sharing [29]. For instance, Cai et al. [82] reused weights of previously discovered networks to amortize the training cost. Pham et al. [29] shared parameters between different child networks to improve the efficiency of the searching procedure. Brock et al. [55] utilized a network to generate model parameters given a discovered network, avoiding fully training from scratch. Liu et al. [28] relaxed the search space to be continuous, so that they can use gradient descent to effectively search cells. Though these approaches successfully accelerate the architecture search procedure, several GPU days are still required [70, 82]. Our work GDAS [11] samples individual architecture in a differentiable way to effectively discover architecture.

2.2 Hyperparameter Optimization (HPO)

Black-box and multi-fidelity HPO methods have a long standing history [24, 84, 85, 31, 86, 31]. Black-box methods, e.g., grid search and random search [24], regard the evaluation function as a black-box. They sample some hyperparameters and evaluate them one by one to find the best. Bayesian methods can make the sampling procedure in random search more efficient [87, 25, 88]. They employ a surrogate model and an acquisition function to decide which candidate to evaluate next [89]. Multi-fidelity optimization methods accelerate the above methods by evaluating on a proxy task, e.g., using less training epochs or a subset of data [44, 90, 86, 91]. These HPO methods are computationally expensive to search for deep learning models [16].

Recently, gradient-based HPO methods have shown better efficiency [92, 48], by computing the gradient with respect to the hyperparameters. For example, Maclaurin et al. [43] calculate the extract gradients w.r.t. hyperparameters. Fabian [93]

leverages the implicit function theorem to calculate approximate hyper-gradient. Following that, different approximation methods have been proposed [48, 93, 94]. Despite of their efficiency, they can only be applied to differentiable hyperparameters such as weight decay, but not non-differentiable hyperparameters, such as learning rate [48] or optimizer [94].

Few approaches have been developed for the joint searching of hyperparameters and architectures [95, 96]. However, they focus on small datasets and small search spaces, and are computationally expensive.

2.3 AutoDL Benchmarks

In the past few years, different kinds of search spaces and search algorithms have been proposed. They brought great advancements in many applications of neural network, such as visual perception [33, 97, 37], language modelling [29, 28, 11], etc. Despite their success, many researchers have raised concerns about the reproducibility and generalization ability of the NAS algorithms [34, 98, 6, 79, 79, 99]. It is essentially not clear if the reported improvements have come from hyperparameter settings, re-training pipelines, random seeds, or the improvements of the searching algorithm itself [34].

To the best of our knowledge, NAS-Bench-101 [98] is the first large-scale architecture dataset. NAS-Bench-101 transforms the problem of architecture search into the problem of searching neural cells, represented as a DAG. Different from ours (Chapter 5), NAS-Bench-101 defines operation candidates on the node, whereas we associate operations on the edge as inspired by [28, 11, 54]. The main highlights of our benchmark are as follows. (1) We are algorithm-agnostic while NAS-Bench-101 without any modification is only applicable to selected algorithms [100, 99]. The original complete search space, based on the nodes in NAS-Bench-101, is huge. So, it is exceedingly difficult to efficiently traverse the training of all architectures. To

trade off the computational cost and the size of the search space, they constrain the maximum number of edges in the DAG. However, it is difficult to incorporate this constraint in all NAS algorithms, such as NAS algorithms based on parameter sharing [28, 29]. Therefore, many NAS algorithms cannot be directly evaluated on NAS-Bench-101. Our benchmark solves this problem by sacrificing the number of nodes and including all possible edges so that our search space is algorithm-agnostic. (2) We provide extra diagnostic information, such as architecture computational cost, fine-grained training and evaluation time, etc., which we hope will give inspirations to better and most efficient designs of NAS algorithms utilizing these diagnostic information.

Despite the existence of NAS-Bench-101, other researchers have also devoted their effort to building a fair comparison and development environments for NAS. Zela et al. [99] proposed a general framework for one-shot NAS methods and reused NAS-Bench-101 to benchmark different NAS algorithms. Yu et al. [101] designed a novel evaluation framework to evaluate the search phase of NAS algorithms by comparing with a random search. Nikita et al. [102] proposed an RNN-based architecture benchmark for the NLP task. The aforementioned works have mainly focused on the network topology but as other aspects of DNNs, such as network size and optimizer, significantly affect the network's performance there is a need for an environment and systematic studies covering these areas of NAS. Unfortunately, until now these aspects have rarely been considered w.r.t. the problem of reproducibility and generalization ability.

NAS-HPO-Bench [95] evaluated 62208 configurations in the joint NAS and hyperparameter space for a simple 2-layer feed-forward network. Since NAS-HPO-Bench has only 144 architectures, it may be insufficient to evaluate different NAS algorithms. The NAS-HPO-Bench dataset also includes the number of channels in a multilayer perceptron (MLP). In contrast, our benchmark has a much larger size

search space than NAS-HPO-Bench and provides the useful information on deep architecture instead of shallow MLP.

In addition to the above discussion, we summarize the unique aspect of different NAS benchmarks in Table 2.2.

Name	Publicly Available	#Datasets	search space	#Config	Application
NAS-Bench-101 [98]	2019.02	1	topology	423K	image recognition
NAS-HPO-Bench [95]	2019.05	4	shallow network + HP	62K	four UCI tasks
NAS-Bench-201 [6]	2020.01	3	topology	15.6K	image recognition
NAS-Bench-NLP [102]	2020.06	1	topology	14.3K	language modeling
NAS-Bench-301 [103]	2020.08	1	topology	10^{18}	image recognition
NATS-Bench [1]	2020.09	3	topology + size	15.6K + 32.8K	image recognition
HW-NAS-Bench [104]	2021.01	3	topology	10^{21}	hardware metric
NAS-Bench-ASR [105]	2021.01	1	topology	8K	speech recognition

Table 2.2 : We compare the unique aspect of different NAS benchmarks.

2.4 AutoDL Software

Software frameworks have greatly influenced and fueled the advancement of machine learning. The need for computing gradients has made auto-gradient based frameworks [106, 107, 108, 109, 110, 111] flourish. To support modular machine learning programs with the flexibility to modify them, frameworks were introduced with an emphasis on hyper-parameter management [112, 113]. The sensitivity of machine learning to hyper-parameters and model architecture has led to the advent of AutoML libraries [114, 115, 116, 117, 118, 119, 120, 121, 122]. Some (e.g., [114, 115, 116]) formulate AutoML as a problem of jointly optimizing architectures and hyper-parameters. Others (e.g., [117, 118, 119]) focus on providing interfaces for black-box optimization. In particular, Google's Vizier library [117] provides tools for optimizing a user-specified search space using black-box algorithms [24, 90], but makes the end user responsible for translating a point in the search space into

a user program. DeepArchitect [120] proposes a language to create a search space as a program that connects user components. Keras-tuner [121] employs a different way to annotate a model into a search space, though this annotation is limited to a list of supported components. Optuna [123] embraces eager evaluation of tunable parameters, making it easy to declare a search space on the go (Appendix B.4). Meanwhile, efficient NAS algorithms [29, 28, 72] brought new challenges to AutoML frameworks, which require coupling between the controller and child program. AutoGluon [119] and NNI [118] partially solve this problem by building predefined modules that work in both general search mode and weight-sharing mode, however, supporting different efficient NAS algorithms are still non-trivial. Among the existing AutoML systems, complex search flows are less explored. Compared to them, PyGlove [7] employs a mutable programming model to solve these problems, making AutoML easily accessible to preexisting ML programs. It also accommodates the dynamic interactions among the child programs, search spaces, search algorithms, and search flows to provide the flexibility needed for future AutoML research.

2.5 Summary and Discussion

In this section, we critically discuss the topics of efficiency, generalizability, and reproducibility of the AutoDL algorithms. Later, we will introduce the motivations of our choices and other promising directions.

Efficiency The major computational bottleneck of AutoDL algorithms is the high computational resources of evaluating a single deep learning program and a large number of programs to be evaluated. Therefore, how to reduce a single program's cost and the number of programs are the focus of efficient AutoDL algorithms. Moreover, an efficient AutoDL algorithm must consider both of these aspects. Inherited from AutoML algorithms, an important direction is to use a proxy program

to produce a low-fidelity approximation for the target program. For example, we can approximate the accuracy of a CNN by training it only for five epochs [60]. Another popular direction that aligns well with deep learning is parameter sharing (or weight sharing, as it is referred to in some literature). The parameter sharing approach can reuse the parameters of a previously trained model, and consequently, the cost of a single deep learning program can be significantly reduced by avoiding training from scratch [29, 48]. On average, such cost can be reduced to a level similar to a single forward iteration. The third direction is to predict the performance of a deep learning program with a learnable function. For example, a neural predictor is proposed to predict the validation accuracy by using the architecture configuration as its input [124].

Among these three directions, the first one still requires massive computational cost, because the cost of one epoch for deep learning is still expensive. The second one can obtain quite good parameters for each candidate, though it requires customized designs for different scenarios. The third one is as flexible as the first, while collecting the training data for the performance predictor is not affordable for everyone. In sum, each direction has its own advantages and disadvantages.

Regarding how to minimize the programs to be evaluated, the most commonly used approach is a Bayesian optimization, and some of the other approaches include RL and EA. For these approaches, the number of evaluated programs and their performance is a trade-off. The more programs that they see, the higher their performance usually gets. However, they could be less capable of handling a large-scale search space. Apart from these, hypergradient-based approaches stand out due to their scalability. They try to compute the gradient of hyperparameters with regard to the validation performance [48, 28, 5]. With such gradients, even millions of hyperparameters can be simultaneously optimized.

In this thesis, we mainly concentrate on the differentiable AutoDL due to the following reasons. Firstly, it implicitly leverages the parameter sharing techniques and is much more efficient than designing a proxy program or learning a performance predictor. Secondly, differentiable AutoDL can optimize the distribution of architectures or hyperparameters via gradient descent. It can be more easily scaled up to a large search space and tested over time with many theoretical supports. Apart from these benefits, differentiable AutoDL is the white box algorithm, and thus requires the user to provide the detailed formulation of the target deep learning program. As a result, when switching from one search space to another one, customized modification is desired. In addition, differentiable AutoDL can not handle a deep learning program with non-differentiable operations. These two problems the key restrictions of its applicability.

Generalizability Considering whether the specific formulation of a deep learning program is required or not, the AutoDL algorithms can be categorized into black-box and white-box approaches. Black-box approaches such as random search [24], Bayesian optimization [25], and RL [27] can be applied to any kind of search space, especially the search space with unknown candidates. Such flexibility sacrifices efficiency since they must execute many candidate deep learning programs in the search space. In contrast, white-box approaches, such as hypergradient method [48] and ENAS [29], leverage the intrinsic and prior knowledge of a specific search space, and thus usually can enable better efficiency and higher performance. For example, ENAS [29] assumes the output shape of different candidate operations is the same. IFT [48] must be applied to differentiable and continuous hyperparameters. In this thesis, we choose to focus on and further investigate the differentiable AutoDL belonging to the white-box class of approaches. A general differentiable AutoDL is proposed in Chapter 4 to preserve the benefits of the white-box approach while

absorbing some of the generalizability aspects of the black-box approach.

Reproducibility Reproducing the results of machine learning and especially deep learning algorithms is a painstaking work. Many researchers have pointed out the reproducibility crisis in machine learning (or general artificial intelligence). Apart from not publishing codes, a major issue is non-determinism, e.g., random initialization, data shuffling, randomness in optimization and neural networks, different frameworks, different environments, etc. As the reproducibility problem has been extensively discussed before, here, we would focus on the reproducibility in AutoDL.

AutoDL is naturally more complex than deep learning, and it is more challenging to make AutoDL reproducible. The paper of [34] discussed how to scientifically research on NAS, where many concepts can be applied to the general AutoDL[†]. Different from reproducible machine learning, one important aspect in AutoDL is how to report the performance of the final discovered deep learning program. Different hyperparameters can cause significant performance gap for this deep learning program; different frameworks or environments might also cause a slight performance change; running the same code twice is also possible to get two different results. To solve this problem, we have built NATS-Bench providing off-the-shelf architecture performance on three datasets [1]. With the help of this benchmark, users can directly query the performance of a deep learning program, i.e., an architecture candidate running on one dataset. Therefore, for the same deep learning program, different users will get the same performance. There is still a long way to go for reproducible AutoDL, and we just made an early effort towards this direction.

[†]It is highly recommended to read [34] if one is interested in studying NAS.

Chapter 3

Differentiable Neural Architecture Search

3.1 Introduction

In this chapter, we study how to accelerate the NAS algorithms, aiming to answer the first question described in Chapter 1 – (Q-1 "How could we reduce the massive computational cost of AutoDL algorithms?"). We proposed a series of coherent solutions to achieve the objectives – Obj-2, Obj-3, and Obj-4, which are identified in Chapter 1.

The neural architecture is usually determined by two aspects: architecture topology and architecture size. Some works [125] use topology to indicate the connectivity pattern of architecture. In this chapter, the terminology "architecture topology" or "topology" refers to the connection topology and the associated operation on each connection. The architecture size indicates the depth and width of a neural architecture, where the width is the number of channels in each layer. It usually requires a customized design to accelerate the search of different architecture aspects. Thus, we will introduce how to enable efficient search for topology in Section 3.2 and size in Section 3.3.

3.2 Efficient NAS using A Differentiable Sampler

We propose a **G**radient-based searching approach using **D**ifferentiable **A**rchitecture **S**ampling (GDAS). It can search for a robust neural architecture in four hours with a single V100 GPU. GDAS significantly improves efficiency compared to the previous methods. We start by searching for a robust neural "cell" instead of a neural

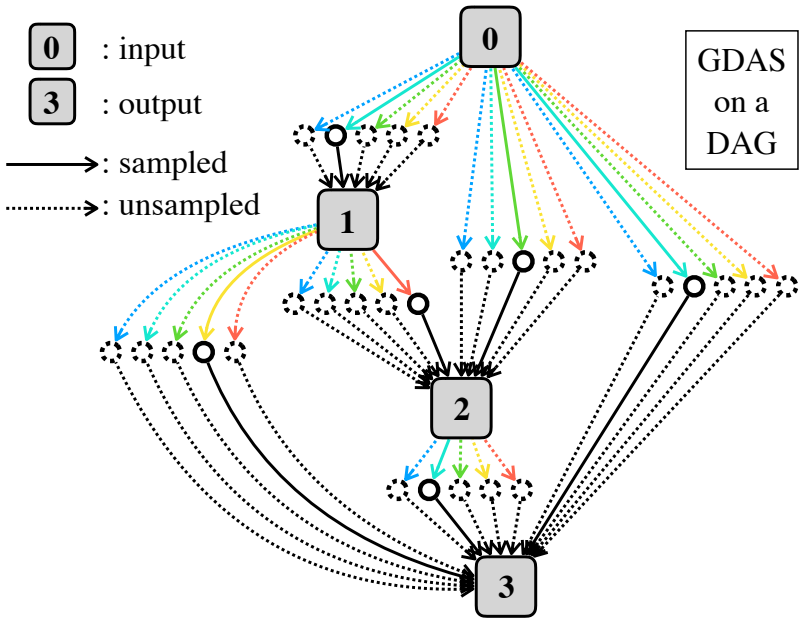


Figure 3.1 : The search space of a neural cell is represented by a DAG.

network [33, 54]. A neural cell contains multiple functions to transform features, and a neural network consists of many copies of the discovered neural cell [70, 54]. Figure 3.1 illustrates our searching procedure in detail. We represent the search space of a cell by a DAG. Every grey square node indicates a feature tensor, numbered by the computation order. Different colored arrows indicate different kinds of operations, which transform one node into its intermediate features. Meanwhile, each node is the sum of the intermediate features transformed from the previous nodes. During training, the proposed GDAS samples a sub-graph from the whole DAG, indicated by solid connections in Figure 3.1. In this sub-graph, each node only receives one intermediate feature from every previous node. Specifically, among the intermediate features between every two nodes, GDAS samples one feature in a differentiable way. In this way, GDAS can be trained by gradient descent to discover a robust neural cell in an end-to-end fashion.

The fast searching ability of GDAS is mainly due to the sampling behavior.

A DAG contains hundreds of parametric operations with millions of parameters. Directly optimizing this DAG [28] instead of sampling a sub-graph leads to two disadvantages. First, it costs a lot of time to update numerous parameters in one training iteration, increasing the overall training time to more than one day [28]. Second, optimizing different operations together could make them compete with each other. For example, different operations could generate opposite values. The sum of these opposite values tends to vanish, breaking the information flow between the two connected nodes and destabilizing the optimization procedure. To solve these two problems, the proposed GDAS samples a sub-graph at one training iteration. As a result, we only need to optimize a part of the DAG at one iteration, which accelerates the training procedure. Moreover, inappropriate competition is avoided, which makes the optimization effective.

In summary, GDAS has the following benefits:

1. Compared to previous RL-based and ES-based methods, GDAS makes the searching procedure differentiable, which allows us to end-to-end learn a robust searching rule by gradient descent. For RL-based and ES-based methods, feedback (reward) is obtained after a prolonged training trajectory, while feedback (loss) in our gradient-based method is instant and is given in every iteration. As a result, the optimization of GDAS is potentially more efficient.

2. Instead of using the whole DAG, GDAS samples one sub-graph at one training iteration, accelerating the searching procedure. Besides, the sampling in GDAS is learnable and contributes to finding a better cell.

3. GDAS delivers a strong empirical performance while using fewer GPU resources. On CIFAR-10, GDAS can finish one searching procedure in several GPU hours and discover a robust neural network with a test error of 2.82%. On PTB, GDAS discovers an RNN model with a test perplexity of 57.5. Moreover, the net-

works discovered on CIFAR and PTB can be successfully transferred to ImageNet and WT2.

3.2.1 Methodology

Search Space as a DAG

We search for the neural cell in the search space and stack this cell in series to compose the whole neural network. For CNN, a cell is a fully convolutional network that takes output tensors of previous cells as inputs and generates another feature tensor. For recurrent neural network (RNN), a cell takes the feature vector of the current step and the hidden state of the previous step as inputs, and generates the current hidden state. For simplification, we take CNN as an example for the following description.

We represent the cell in CNN as a DAG $\mathcal{G}$ consisting of an ordered sequence of B computational nodes. Each computational node represents one feature tensor, which is transformed from two previous feature tensors. This procedure can be formulated as shown in Eq. (3.1) following [54].

$$\boldsymbol{I}_i = f_{i,j}(\boldsymbol{I}_j) + f_{i,k}(\boldsymbol{I}_k) \quad \text{s.t.} \quad j < i \ \ \& \ \ k < i, \tag{3.1}$$

where $\boldsymbol{I}_i$, $\boldsymbol{I}_j$, and $\boldsymbol{I}_k$ indicate the i-th, j-th, and k-th nodes, respectively. $f_{i,j}$ and $f_{i,k}$ indicate two functions from the candidate function set $\mathbb{F}$. We denote the computational nodes of a cell as B. Taking $B = 4$ as an example, a cell contains 7 nodes in total, i.e., $\{\boldsymbol{I}_i | 1 \leq i \leq 7\}$. $\boldsymbol{I}_1$ and $\boldsymbol{I}_2$ nodes are the cell outputs in the previous two layers. $\boldsymbol{I}_3$, $\boldsymbol{I}_4$, $\boldsymbol{I}_5$, and $\boldsymbol{I}_6$ nodes are the computational nodes calculated by Eq. (3.1). $\boldsymbol{I}_7$ indicates the output tensor of this cell, which is the concatenation of the four computational nodes, i.e., $\boldsymbol{I}_7 = \boldsymbol{I}_3 \frown \boldsymbol{I}_4 \frown \boldsymbol{I}_5 \frown \boldsymbol{I}_6$. In GDAS, the candidate function set $\mathbb{F}$ contains the following 8 functions: (1) identity, (2) zeroize, (3) 3x3 depth-wise separate conv, (4) 3x3 dilated depth-wise separate conv, (5) 5x5 depth-wise separate conv, (6) 5x5 dilated depth-wise separate conv, (7) 3x3 average pooling, (8)

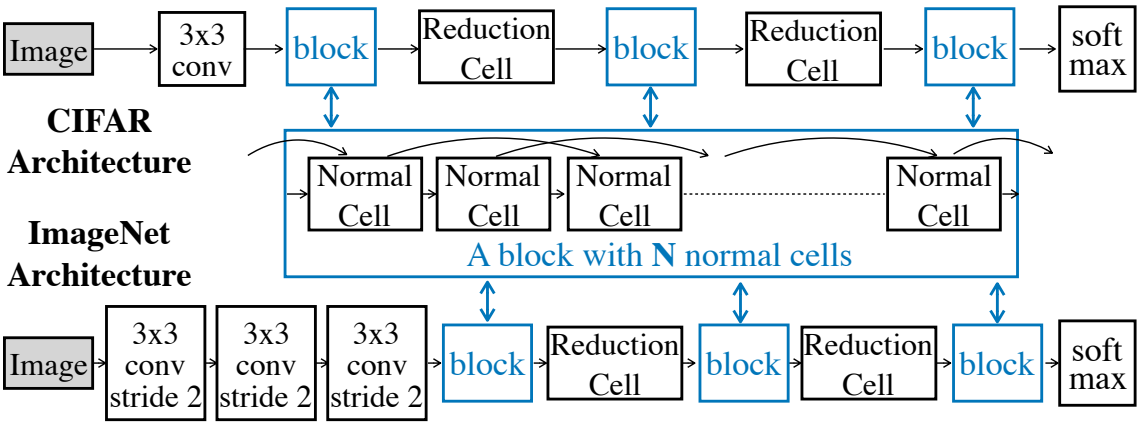


Figure 3.2 : The strategy to design CIFAR and ImageNet architecture.

3x3 max pooling. We use the same candidate function set $\mathbb{F}$ as [28], which is similar to [54] but removes some unused functions and adds some useful functions.

From cell to network. We search for two kinds of cells, i.e., a normal cell and a reduction cell. For the normal cell, each function in $\mathbb{F}$ has the stride of 1. For the reduction cell, each function in $\mathbb{F}$ has the stride of 2. Once we discover one normal cell and one reduction cell, we stack many copies of these discovered cells to make up a neural network. As shown in Figure 3.2, for the CIFAR architecture, we stack N normal cells as one block. Given an image, it first forwards through the network head part, i.e., one 3 by 3 convolutional layer. It then forwards through three blocks with two reduction cells in between. The ImageNet architecture is similar to the CIFAR architecture, but the network head part consists of three 3 by 3 convolutional layers. We follow [28] to setup these two overall structures.

Searching by Differentiable Model Sampling

Recall Eq. (5.1), we denote a neural architecture as α and the weights of this neural architecture as ω_α. In our GDAS, we use the typical classification loss as the

objective function:

$$\mathcal{L}(\alpha, \omega_\alpha, \mathcal{D}_{train}) = \mathbb{E}_{(x,y)\sim\mathcal{D}_{train}} - \log \Pr(y|x; \alpha, \omega_\alpha),$$
$$\mathcal{L}(\alpha, \omega^*_\alpha, \mathcal{D}_{val}) = \mathbb{E}_{(x',y')\sim\mathcal{D}_{val}} - \log \Pr(y'|x'; \alpha, \omega^*_\alpha), \tag{3.2}$$

Remember that $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$ indicate the training set and the validation set, respectively. (x, y) and (x', y') are the data associated with its label, which are sampled from $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$, respectively.

An architecture α consists of many copies of the neural cell. This cell is sampled from the search space represented by $\mathcal{G}$. Specifically, between node$_i$ and node$_j$, we sample one transformation function from $\mathbb{F}$ from a discrete probability distribution $\mathcal{T}_{i,j}$. During the search, we calculate each node in a cell as:

$$\boldsymbol{I}_i = \sum_{j=1}^{i-1} f_{i,j}\ (\boldsymbol{I}_j; \boldsymbol{W}_{f_{i,j}}) \quad \text{s.t.} \quad f_{i,j} \sim \mathcal{T}_{i,j}, \tag{3.3}$$

where $f_{i,j}$ is sampled from $\mathcal{T}_{i,j}$ and $\boldsymbol{W}_{f_{i,j}}$ is its associated weight. The discrete probability distribution $\mathcal{T}_{i,j}$ is characterized by a learnable probability mass function as in Eq. (3.4):

$$\Pr(f_{i,j} = \mathbb{F}_k) = \frac{\exp(\boldsymbol{A}^k_{i,j})}{\sum_{k'=1}^{K} \exp(\boldsymbol{A}^{k'}_{i,j})}, \tag{3.4}$$

where $\boldsymbol{A}^k_{i,j}$ is the k-th element of a K-dimensional learnable vector $\boldsymbol{A}_{i,j} \in \mathbb{R}^K$, and $\mathbb{F}_k$ indicates the k-th function in $\mathbb{F}$. K is the cardinality of $\mathbb{F}$, i.e., $K = |\mathbb{F}|$. Actually, $\boldsymbol{A}_{i,j}$ encodes the sampling distribution of the function between node$_i$ and node$_j$. As a result, the sampling distribution of a neural cell is encoded by all $\boldsymbol{A}_{i,j}$, i.e., $\mathcal{A} = \{\boldsymbol{A}_{i,j}\}$.

Given Eq. (3.3) and Eq. (3.4), we can obtain α and ω, and thus can calculate $\Pr(y|x; \alpha, \omega)$ in Eq. (3.2). However, since Eq. (3.3) needs to sample from a discrete probability distribution, we cannot back-propagate gradients through $\boldsymbol{A}_{i,j}$ in Eq. (3.4) to optimize $\boldsymbol{A}_{i,j}$. To allow back-propagation, we first use the Gumbel-Max

trick [126, 127] to re-formulate Eq. (3.3) as Eq. (3.5), which provides an efficient way to draw samples from a discrete probability distribution.

$$\boldsymbol{I}_i = \sum_{j=1}^{i-1} \sum_{k=1}^{K} \boldsymbol{h}_{i,j}^k \, \mathbb{F}_k(\boldsymbol{I}_j; \boldsymbol{W}_{i,j}^k), \tag{3.5}$$

$$\text{s.t.} \quad \boldsymbol{h}_{i,j} = \text{one\_hot}(\arg\max_k(\boldsymbol{A}_{i,j}^k + \boldsymbol{o}_k)), \tag{3.6}$$

where $\boldsymbol{o}_k$ are i.i.d samples drawn from Gumbel (0,1)[*], and $\boldsymbol{h}_{i,j}^k$ is the k-th element of $\boldsymbol{h}_{i,j}$. $\boldsymbol{W}_{i,j}^k$ is the weight of $\mathbb{F}_k$ for the transformation function between node$_i$ and node$_j$. Then, we use the softmax function to relax the arg max function so as to make Eq. (3.5) being differentiable [65, 128]. Formally, we use Eq. (3.7) to approximate Eq. (3.5).

$$\tilde{\boldsymbol{h}}_{i,j}^k = \frac{\exp((\log(\Pr(f_{i,j} = \mathbb{F}_k)) + \boldsymbol{o}_k)/\tau)}{\sum_{k'=1}^{K} \exp((\log(\Pr(f_{i,j} = \mathbb{F}_{k'})) + \boldsymbol{o}_{k'})/\tau)}, \tag{3.7}$$

where τ is the softmax temperature. When $\tau \to 0$, $\tilde{\boldsymbol{h}}_{i,j}^k = \boldsymbol{h}_{i,j}^k$. When $\tau \to \infty$, each element in $\tilde{\boldsymbol{h}}$ will be the same and the approximated distribution will be smooth. To be noticed, we use the arg max function in Eq. (3.5) during the forward pass but the soft max function in Eq. (3.7) during the backward pass to allow gradient back-propagation.

Training. Reviewing the objective of NAS in Eq. (3.2), the main challenge is learning to find architecture α. By utilizing Eq. (3.7), we can make the sampling procedure differentiable and learn a distribution of neural cells (representing architectures). However, it is still intractable to directly solve Eq. (3.2), because the nested formulation in Eq. (3.2) needs to calculate high order derivatives. In practice, to avoid calculating high order derivatives, we apply the alternative optimization strategy to update the sampling distribution $\mathcal{T}_{\mathcal{A}}$ and the weights of all functions $\mathcal{W}$ in an iterative way. Given one data sample x and its associated label

[*] $\boldsymbol{o}_i = -\log(-\log(u))$ with $u \sim \text{Unif}\ [0, 1]$

Algorithm 1 The GDAS algorithm

Input: split the training set into two disjoint sets: $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$; randomly initialized $\mathcal{A}$ and $\mathcal{W}$, and the batch size n

while not converge **do** ▷ search for an architecture

Sample batch of data $\mathbb{D}_t = \{(x_i, y_i)\}_{i=1}^n$ from $\mathcal{D}_{train}$

Calculate $L_T = \sum_{i=1}^n \ell(x_i, y_i)$ based on Eq. (3.8)

Update $\mathcal{W}$ by gradient descent: $\mathcal{W} = \mathcal{W} - \nabla_{\mathcal{W}} L_T$

Sample batch of data $\mathbb{D}_v = \{(x_i, y_i)\}_{i=1}^n$ from $\mathcal{D}_{val}$

Calculate $L_V = \sum_{i=1}^n \ell(x_i, y_i)$ based on Eq. (3.8)

Update $\mathcal{A}$ by gradient descent: $\mathcal{A} = \mathcal{A} - \nabla_{\mathcal{A}} L_V$

end while

Derive the final architecture from $\mathcal{A}$

Optimize the architecture on the training set

y, we calculate the loss as:

$$\ell(x, y) \quad = -\log \Pr(y|x; \alpha, \omega_\alpha), \tag{3.8}$$

$$\text{s.t.} \quad \alpha \sim \mathcal{T}_{\mathcal{A}} \ \& \ \omega_\alpha \subset \mathcal{W}, \tag{3.9}$$

where $\mathcal{T}_{\mathcal{A}}$ is the distribution encoded by $\mathcal{A}$ and $\mathcal{W} = \{\boldsymbol{W}_{i,j}^k\}$ represents the weights of all functions in all cells of the network. Note that, for one data sample, it first samples α from $\mathcal{T}_{\mathcal{A}}$ and then calculates the network output only on its associated weight ω_α, which is a part of $\mathcal{W}$. As shown in Algorithm 1, we apply the AOS to update $\mathcal{A}$ based on the validation losses from $\mathcal{D}_{val}$ and update $\mathcal{W}$ based on the training losses from $\mathcal{D}_{train}$. It is essential to train $\mathcal{W}$ on $\mathcal{D}_{train}$ and $\mathcal{A}$ on $\mathbb{D}_V$, because (1) this strategy is theoretically sound with the objective Eq. (3.2); and (2) this strategy can improve the generalization ability of the searched structure.

Architecture. After training, we need to derive the final architecture from the

learned $\mathcal{A}$. Each node$_i$ connects with T previous nodes. Following the previous works, we use $T = 2$ for CNN [54, 28] and $T = 1$ for RNN [29, 28]. Suppose Ω is the candidate index set, we derive the final architecture by the following procedure: (1) define the importance of the connection between node$_i$ and node$_j$ as: $\max_{k\in\Omega} \Pr(f_{i,j} = \mathbb{F}_k)$. (2) for each node$_i$, retain T connections with the maximum importance from the previous nodes. (3) for the retained connection between node$_i$ and node$_j$, we use the function $\mathbb{F}_{\arg\max_{k\in\Omega} \Pr(f_{i,j}=\mathbb{F}_k)}$. Ω is $\{1, ..., K\}$ by default.

Acceleration. In Eq. (3.5), $\boldsymbol{h}_{i,j}$ is a one-hot vector. As a result, in the forward procedure, we only need to calculate the function $\mathbb{F}_{\arg\max(\boldsymbol{h}_{i,j})}$. During the backward procedure, we only back-propagate the gradient generated at the $\arg\max(\tilde{\boldsymbol{h}}_{i,j})$. In this way, we can save most computation time and also reduce the GPU memory cost by about $|\mathbb{F}|$ times. Within one training batch, a different cell will produces a different $\boldsymbol{h}_{i,j}$, which was shared for each training examples.

One benefit of this acceleration trick is that it allows us to directly search on the large-scale dataset (e.g., ImageNet) due to the saved GPU memory. We did some experiments to directly search on ImageNet using the same hyperparameters as on the small datasets, however, failed to obtain a good performance. Searching on a large-scale dataset might require different hyperparameters and needs careful tuning. We will explore this in our future work.

3.2.2 Experimental Study

We evaluate our GDAS on **CIFAR-10**, **CIFAR-100** [132], **ImageNet** [133], **Penn Treebank (PTB)** [134], and **WikiText-2 (WT2)** [135].

Search for CNN

CNN Searching Setup. The neural cells for CNN are searched on CIFAR-10 following [33, 54, 70, 29]. We randomly split the official training images into two

Type	**Method**	**Venue**	**GPUs**	**Times** (days)	**Parameters** (million)	**Error** on CIFAR-10	**Error** on CIFAR-100
Human Expert	ResNet + CutOut [20]	CVPR16	–	–	1.7	4.61	22.10
	DenseNet-BC [129]	CVPR17	–	–	25.6	3.46	17.18
Macro Search Space	MetaQNN [45]	ICLR17	10	8-10	11.2	6.92	27.14
	Net Transformation [82]	AAAI18	5	2	19.7	5.70	–
	SMASH [55]	ICLR18	1	1.5	16.0	4.03	–
	NAS [33]	ICLR17	800	21-28	7.1	4.47	–
	NAS + more filters [33]	ICLR17	800	21-28	37.4	3.65	–
	ENAS [29]	ICML18	1	0.32	38.0	3.87	–
Micro Search Space	Hierarchical NAS [69]	ICLR18	200	1.5	61.3	3.63	–
	Progressive NAS [70]	ECCV18	100	1.5	3.2	3.63	19.53
	NASNet-A [54]	CVPR18	450	3-4	3.3	3.41	–
	NASNet-A + CutOut [54]	CVPR18	450	3-4	3.3	**2.65**	–
	ENAS [29]	ICML18	1	0.45	4.6	3.54	19.43
	ENAS + CutOut [29]	ICML18	1	0.45	4.6	2.89	–
	DARTS (1st) + CutOut [28]	ICLR19	1	0.38	3.3	3.00	–
	DARTS (2nd) + CutOut [28]	ICLR19	1	1	3.4	2.82	**17.54**†
	GHN + CutOut [130]	ICLR19	1	0.84	5.7	2.84	–
	NAONet [131]	NeurIPS18	200	1	10.6	3.18	–
	AmoebaNet-A + CutOut [76]	AAAI19	450	7	3.1	3.12	18.93†
	GDAS [C=36,N=6]	CVPR19	1	0.21	3.4	3.87	19.68
	GDAS [C=36,N=6] + CutOut	CVPR19	1	0.21	3.4	2.93	18.38

Table 3.1 : Classification errors of GDAS and baselines on CIFAR.

groups, with each group containing 25K images. One group is used as the training set $\mathbb{D}_T$ in Algorithm 1, and the other is used as the validation set $\mathbb{D}_V$ in Algorithm 1. The candidate function set $\mathbb{F}$ has 8 different functions as introduced in Section 3.2.1. The default hyperparameters for each function in $\mathbb{F}$ are the same in [28, 54, 70]. By default, we set the number of initial channels in the first convolution layer C as 16; set the number of computational nodes in a cell B as 4; and the number of layers in one block N as 2. We train the model by 240 epochs in total. For ω, we use the SGD optimization. We start with a learning rate of 0.025 and anneal it down

to 1e-3 following a cosine schedule. We use the momentum of 0.9 and the weight decay of 3e-4. For α, we use the Adam optimization [136] with the learning rate of 3e-4 and the weight decay of 1e-3. The τ is initialized as 10 and is linearly reduced to 0.1. To search the normal cell and the reduction cell on CIFAR-10, our GDAS takes about five hours to finish the search procedure on a single NVIDIA Tesla V100 GPU. *Following [28], we run GDAS 4 times with different random seeds and pick the best cell based on its validation performance. This procedure can reduce the high variance of the searched results, especially when searching the RNN structure.*

Clarifications on the searching cost (GPU days) of different methods. The searching costs listed in Table 3.1 and Table 3.2 are **not normalized** across different GPU devices. Different algorithms might run on different machines, and we simply refer the searching costs reported in their papers.

Discussion on the acceleration step. If we do not apply the acceleration step introduced in Section 3.2.1, each iteration will cost $|\mathbb{F}|$=8$\times$ more time and GPU memory than GDAS. In the same time, without this acceleration step, it requires less training epochs to converge but still costs more time than applying the acceleration step.

Results on CIFAR. After the searching procedure, we use C=36, B=4, and N=6 to form a CNN. Following the previous works [28, 29, 54], we train the network by 600 epochs in total. We start the learning rate of 0.025 and reduce it to 0 with the cosine learning rate scheduler. We set the probability of path dropout as 0.2 and the auxiliary tower with the weight of 0.4 [33]. We use the standard pre-processing and data augmentation, i.e., randomly cropping, horizontally flipping, normalization, and CutOut [139].

We compare the models discovered by our approach with other state-of-the-art models in Table 3.1. The models discovered by the macro search algorithms obtain

Type	Method	Venue	GPUs	Times (days)	Test Error (%) Top-1	Test Error (%) Top-5	Parameters (million)	+× (million)
Human Expert	Inception-v1 [137]	CVPR15	–	–	30.2	10.1	6.6	1448
	MobileNet-V2 [15]	CVPR18	–	–	28.0	–	3.4	300
	ShuffleNet [138]	CVPR18	–	–	26.3	–	∼5	524
Micro Search Space	Progressive NAS [70]	ECCV18	100	1.5	25.8	8.1	5.1	588
	NASNet-A [54]	CVPR18	450	3-4	26.0	8.4	5.3	564
	NASNet-B [54]	CVPR18	450	3-4	27.2	8.7	5.3	488
	NASNet-C [54]	CVPR18	450	3-4	27.5	9.0	4.9	558
	DARTS (2nd) [28]	ICLR19	1	1	26.9	9.0	4.9	595
	GHN [130]	ICLR19	1	0.84	27.0	8.7	6.1	569
	AmoebaNet-A [76]	AAAI19	450	7	25.5	8.0	5.1	555
	AmoebaNet-B [76]	AAAI19	450	7	26.0	8.5	5.3	555
	AmoebaNet-C [76]	AAAI19	450	7	**24.3**	**7.6**	6.4	570
	GDAS [C=50,N=4]	CVPR19	1	0.21	26.0	8.5	5.3	581

Table 3.2 : Top-1 and top-5 errors of GDAS and baselines on ImageNet.

a higher error than the models discovered by the micro search algorithms. Using GDAS, we discover a model with 3.3M parameters, which achieves 2.93% error on CIFAR-10. Using GDAS (FRC), we discover a model with only 2.5M parameters, which achieves 2.82% error on CIFAR-10. NASNet-A achieves a lower error rate than ours, but it contains more than 80% of the parameters than the model discovered by GDAS (FRC). Notably, our GDAS discovers a comparable model with the state-of-the-art, whereas the searching cost of our approach is much less than the others. For example, GDAS (FRC) takes less than 4 hours on a single V100 GPU, which is about 0.17 GPU days. It is faster than NASNet by almost 10^4 times. ENAS is a recent work that focuses on accelerating the searching procedure. ENAS is very efficient, whereas our GDAS (FRC) is three times faster than ENAS.

Results on ImageNet. Following [54, 28, 29, 15], we use the ImageNet-mobile setting, in which the input size is 224×224 and the number of multiply-add opera-

Architecture	Perplexity		Parameters	Search Cost
	val	test	(Million)	(GPU days)
V-RHN [140]	67.9	65.4	23	–
LSTM [141]	60.7	58.8	24	–
LSTM + SC [141]	60.9	58.3	24	–
LSTM + SE [142]	58.1	56.0	22	–
NAS [33]	–	64.0	25	10^4
ENAS [29]	60.8	58.6	24	0.5
DARTS (1st) [28]	60.2	57.6	23	**0.13**
DARTS (2nd) [28]	**58.1**	**55.7**	**23**	0.25
GDAS	59.8	57.5	**23**	0.4

Table 3.3 : Comparing the perplexity of different language models on PTB.

tions is restricted to be less than 600M. We train models by SGD with 250 epochs and use the batch size of 128. We initialize the learning rate of 0.1 and reduce it by 0.97 after each epoch.

We compare our results on ImageNet with the other methods in Table 3.2. Most algorithms in Table 3.2 take more than 1000 GPU days to discover a good CNN cell. DARTS [28] uses minimum resources among the compared algorithms, whereas ours is even faster than DARTS [28] by more than 10 times. For GDAS, if we use C=52 and N=4, the number of multiply-add operations will be larger than 600 MB, and thus we use C=50 to restrict it to be less than 600MB. AmoebaNet-A and Progressive NAS achieve a slightly lower test error than ours. However, their methods cost a prohibitive amount of GPU resources. The results in Table 3.2 show the discovered cell on CIFAR-10 can be successfully transferred to ImageNet and

achieve competitive performance.

Search for RNN

Architecture	Perplexity		Parameters	Search Cost
	val	test	(M)	(GPU days)
LSTM + AL [143]	91.5	82.0	28	–
LSTM [141]	69.1	65.9	33	–
LSTM + SC [141]	69.1	65.9	23	–
LSTM + SE [142]	66.0	63.3	33	–
ENAS [29]	72.4	70.4	**33**	0.5
DARTS (2nd) [28]	71.2	69.6	**33**	**0.25**
GDAS	**71.0**	**69.4**	**33**	0.4

Table 3.4 : Comparison with different language models on WT2.

RNN Searching Setup. The neural cells for RNN are searched on PTB with the splits following [28, 29] The candidate function set $\mathbb{F}$ contain 5 functions, i.e., zeroize, Tanh, ReLU, sigmoid, and identity. We use B=9 to search the RNN cell. The RNN model consists of one word embedding layer with a hidden size of 300, one RNN cell with a hidden size of 300, and one decoder layer. We train the model by 200 epochs with a batch size of 128 and a BPTT length of 35. We optimize ω by Adam with a learning rate of 20 and a weight decay of 5e-7. We optimize α by Adam with a learning rate of 3e-3 and a weight decay of 1e-3. Other setups are the same in [29, 28].

Results on PTB. We evaluate the RNN model formed by the discovered recurrent cell on PTB. We use a batch size of 64 and a hidden size of 850. We train the model using the A-SGD by 2000 epochs. The learning rate is fixed as 20 and

the weight decay is 8e-7. DARTS [28] and ENAS [29] greatly reduce the search cost compared to previous methods. Our GDAS incurs a lower search cost than all the previous methods. Note that our code is not heavily optimized and the theoretical search cost should be less than the one reported in Table 3.3.

We compare different RNN models in Table 3.3. The model discovered by GDAS achieves a validation perplexity of 59.8 and a test perplexity of 57.5. The performance of our discovered RNN is on par with the state-of-the-art models in Table 3.3. LSTM + SE [142] obtains better results than ours, but it is an ensemble method using mixture of softmax. By applying the SE technique [142], GDAS can achieve the lower perplexity without doubt. LSTM [141] is an extensively tuned model, whereas our automatically discovered model is superior to it. Compared to other efficient approaches, the search cost of GDAS is the lowest.

Results on WT2. To train the model on WT2, we use the same experiment settings as PTB, but we use a hidden size of 700 and a weight decay of 5e-7. We train the model in 3000 epochs in total. Table 3.4 compares different RNN models on WT2. Our approach achieves competitive results among all automatically searching approaches. GDAS is worse than "LSTM + SC" [141]. Since our model is searched on a small dataset PTB, and the transferable ability of the discovered model might be a little bit weak. If we directly search the RNN model on WT2, we could obtain a better model and improve the transferable ability.

3.2.3 Discussion

Most recent NAS approaches search neural networks on the small-scale datasets, such as CIFAR, and then transfer the discovered networks to the large-scale datasets, such as ImageNet. The obstacle of directly searching on ImageNet is the huge computational cost. GDAS is an efficient NAS algorithm and gives us an opportunity to search on ImageNet. We will explore this research direction in our future work.

3.3 Transformable Architecture Search for Large-scale Data

Deep CNNs have become wider and deeper to achieve high performance on different applications [20, 129, 33]. Despite their great success, it is impractical to deploy them to resource constrained devices, such as mobile devices and drones. A straightforward solution to address this problem is using network pruning [144, 145, 146, 147, 148] to reduce the computation cost of over-parameterized CNNs. A typical pipeline for network pruning, as indicated in Figure 3.3(a), is achieved by removing the redundant filters and then fine-tuning the slashed networks, based on the original networks. Different criteria for the importance of the filters are applied, such as L2-norm of the filter [149], reconstruction error [147], and learnable scaling factor [150]. Lastly, researchers apply various fine-tuning strategies [149, 148] for the pruned network to efficiently transfer the parameters of the unpruned networks and maximize the performance of the pruned networks.

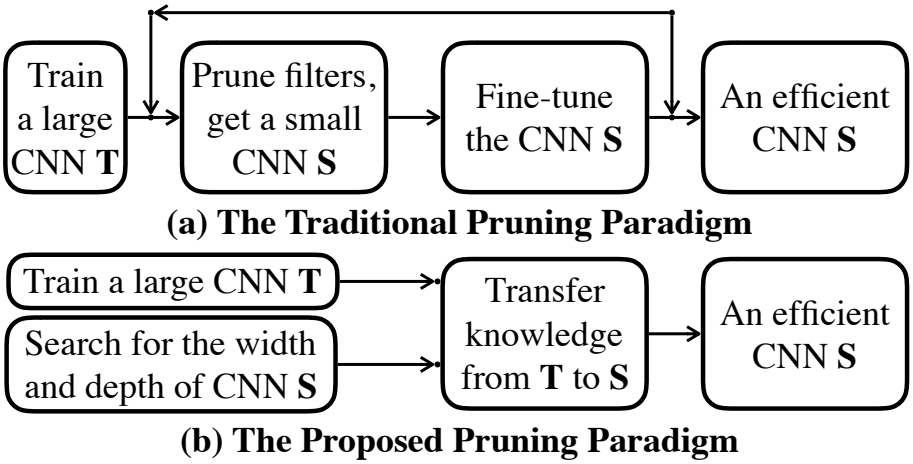


Figure 3.3 : A comparison between the typical pruning paradigm and the proposed paradigm.

Traditional network pruning approaches achieve effective impacts on network compacting while maintaining accuracy. Their network structure is intuitively de-

signed, e.g., pruning 30% filters in each layer [149, 148], predicting sparsity ratio [151] or leveraging regularization [152]. The accuracy of the pruned network is upper bounded by the hand-crafted structures or rules for structures. To break this limitation, we apply NAS to turn the design of the architecture structure into a learning procedure and propose a new paradigm for network pruning as explained in Figure 3.3(b).

Prevailing NAS methods [28, 33, 11, 72, 76] optimize the network topology, while the focus of this chapter is automated network size. In order to satisfy the requirements and make a fair comparison between the previous pruning strategies, we propose a new NAS scheme termed Transformable Architecture Search (TAS). TAS aims to search for the best size of a network instead of topology, regularized by minimization of the computation cost, e.g., FLOPs. The parameters of the searched/pruned networks are then learned by knowledge transfer [153, 154].

TAS is a differentiable searching algorithm, which can search for the width and depth of the networks effectively and efficiently. Specifically, different candidates of channels/layers are attached with a learnable probability. The probability distribution is learned by back-propagating the loss generated by the pruned networks, whose feature map is an aggregation of K feature map fragments (outputs of networks in different sizes) sampled based on the probability distribution. These feature maps of different channel sizes are aggregated with the help of channel-wise interpolation. The maximum probability for the size in each distribution serves as the width and depth of the pruned network.

In experiments, we show that the searched architecture with parameters transferred by knowledge distillation (KD) outperforms previous state-of-the-art pruning methods on CIFAR-10, CIFAR-100 and ImageNet. We also test different knowledge transfer approaches on architectures generated by traditional hand-crafted pruning

approaches [149, 148] and random architecture search approach [28]. Consistent improvements on different architectures demonstrate the generality of knowledge transfer.

3.3.1 Difference between TAS and Pruning Methods

Network pruning [144, 155] is an effective technique to compress and accelerate CNNs, and thus allows us to deploy efficient networks on hardware devices with limited storage and computation resources. A variety of techniques have been proposed, such as low-rank decomposition [156], weight pruning [157, 144, 146, 145], channel pruning [148, 155], dynamic computation [158, 14] and quantization [159, 160]. They lie in two modalities: unstructured pruning [144, 158, 14, 145] and structured pruning [149, 147, 148, 155].

Unstructured pruning methods [144, 158, 14, 145] usually enforce the convolutional weights [144, 157] or feature maps [14, 158] to be sparse. The pioneers of unstructured pruning, LeCun et al. [144] and Hassibi et al. [157], investigated the use of the second-derivative information to prune weights of shallow CNNs. After deep network was born in 2012 [16], Han et al. [145, 146, 161] proposed a series of works to obtain highly compressed deep CNNs based on L2 regularization. After this development, many researchers explored different regularization techniques to improve the sparsity while preserve the accuracy, such as L0 regularization [162] and output sensitivity [163]. Since these unstructured methods make a big network sparse instead of changing the whole structure of the network, they need dedicated design for dependencies [161] and specific hardware to speedup the inference procedure.

Structured pruning methods [149, 147, 148, 155] target the pruning of convolutional filters or whole layers, and thus the pruned networks can be easily developed and applied. Early works in this field [152, 164] leveraged a group Lasso to enable

structured sparsity of deep networks. After that, Li et al. [149] proposed the typical three-stage pruning paradigm (training a large network, pruning, re-training). These pruning algorithms regard filters with a small norm as unimportant and tend to prune them, but this assumption does not hold in deep nonlinear networks [165]. Therefore, many researchers focus on better criterion for the informative filters. For example, Liu et al. [150] leveraged a L1 regularization; Ye et al. [165] applied a ISTA penalty; and He et al. [166] utilized a geometric median-based criterion. In contrast to previous pruning pipelines, our approach allows the number of channels/layers to be explicitly optimized so that the learned structure has high-performance and low-cost.

Besides the criteria for informative filters, the importance of network structure was suggested in [155]. Some methods implicitly find a data-specific architecture [164, 152, 151], by automatically determining the pruning and compression ratio of each layer. In contrast, we explicitly discover the architecture using NAS. Most previous NAS algorithms [33, 11, 28, 76] automatically discover the topology structure of a neural network, while we focus on searching for the depth and width of a neural network. Reinforcement learning (RL)-based [33, 82] methods or evolutionary algorithm-based [76] methods are possible to search networks with flexible width and depth, however, they require huge computational resources and cannot be directly used on large-scale target datasets. Differentiable methods [11, 28, 72] dramatically decrease the computation costs but they usually assume that the number of channels in different searching candidates is the same. TAS is a differentiable NAS method, which is able to efficiently search for a transformable networks with flexible width and depth.

Network transformation [59, 167, 82] also studied the depth and width of networks. Chen et al. [59] manually widen and deepen a network, and proposed Net2Net to initialize the lager network. Ariel et al. [167] proposed a heuristic strategy to

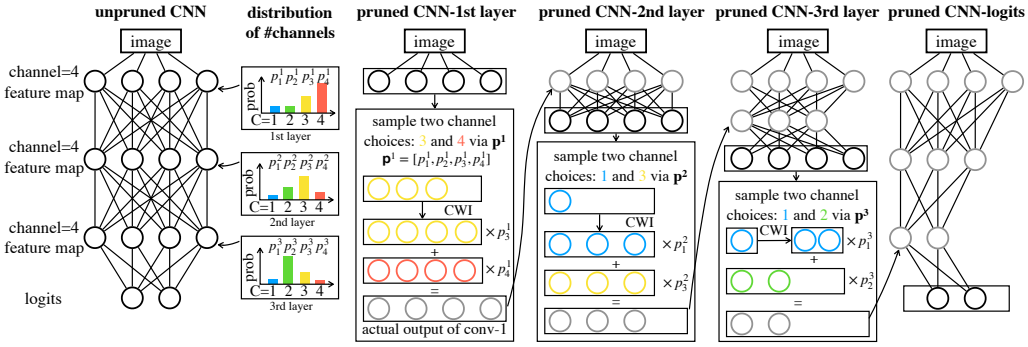


Figure 3.4 : An illustration of TAS on a three-layer CNN.

find a suitable width of networks by alternating between shrinking and expanding. Cai et al. [82] utilized a RL agent to grow the depth and width of CNNs, while our TAS is a differentiable approach and can not only enlarge but also shrink CNNs.

Knowledge transfer has been proven to be effective in the literature of pruning. The parameters of the networks can be transferred from the pre-trained initialization [149, 148]. Minnehan et al. [168] transferred the knowledge of uncompressed network via a block-wise reconstruction loss.

3.3.2 Methodology

Our pruning approach consists of three steps: (1) training the unpruned large network by a standard classification training procedure. (2) searching for the depth and width of a small network via the proposed TAS. (3) transferring the knowledge from the unpruned large network to the searched small network by a simple KD approach [153]. We will introduce the background, show the details of TAS, and explain the knowledge transfer procedure.

Search for width. We use parameters $\widehat{P^w} \in \mathbb{R}^{|\mathbb{C}|}$ to indicate the distribution of the possible number of channels in one layer, indicated by $\mathbb{C}$ and $\max(\mathbb{C}) \leq c_{out}$. The probability of choosing the j-th candidate for the number of channels can be

formulated as:

$$P_j^w = \frac{\exp(\alpha_j)}{\sum_{k=1}^{|\mathbb{C}|}\exp(\alpha_k)} \quad \text{where } 1 \leq j \leq |\mathbb{C}|, \tag{3.10}$$

However, the sampling operation in the above procedure is non-differentiable, which prevents us from back-propagating gradients through P_j^w to $\widehat{P^w}$. Motivated by [11], we apply Gumbel-Softmax [65, 128] to soften the sampling procedure to optimize $\widehat{P^w}$:

$$\widetilde{P^w}_j = \frac{\exp((\log(P_j^w) + \boldsymbol{o}_j)/\tau)}{\sum_{k=1}^{|\mathbb{C}|}\exp((\log(P_k^w) + \boldsymbol{o}_k)/\tau)}, \tag{3.11}$$
$$\text{s.t. } \boldsymbol{o}_j = -\log(-\log(u)) \ \ \& \ \ u \sim \mathcal{U}(0,1),$$

where $\mathcal{U}(0,1)$ means the uniform distribution between 0 and 1. τ is the softmax temperature. When $\tau \to 0$, $\widetilde{P^w}_j$ becomes one-hot, and the Gumbel-softmax distribution drawn from $\widetilde{P^w}_j$ becomes identical to the categorical distribution. When $\tau \to \infty$, the Gumbel-softmax distribution becomes a uniform distribution over $\mathbb{C}$. The feature map in our method is defined as the weighted sum of the original feature map fragments with different sizes, where weights are $\widetilde{P^w}_j$. Feature maps with different sizes are aligned by channel wise interpolation (CWI) so as for the operation of weighted sum. To reduce the memory costs, we select a small subset with indexes $\mathbb{I} \subseteq [|\mathbb{C}|]$ for aggregation instead of using all candidates. Additionally, the weights are re-normalized based on the probability of the selected sizes, which is formulated as:

$$\hat{\boldsymbol{I}} = \sum\nolimits_{j\in\mathbb{I}} \frac{\exp((\log(P_j^w) + \boldsymbol{o}_j)/\tau)}{\sum_{k\in\mathbb{I}}\exp((\log(P_k^w) + \boldsymbol{o}_k)/\tau)} \times \text{CWI}(\boldsymbol{I}_{1:\mathbb{C}_j,:,:}, \max(\mathbb{C}_\mathbb{I})) \ \text{ s.t. } \mathbb{I} \sim \mathcal{T}_{\hat{p}}, \tag{3.12}$$

where $\mathcal{T}_{\hat{p}}$ indicates the multinomial probability distribution parameterized by $\widetilde{P^w}_j$. The proposed CWI is a general operation to align feature maps with different sizes. It can be implemented in many ways, such a 3D variant of spatial transformer

network [169] or adaptive pooling operation [170]. In this section, we choose the 3D adaptive average pooling operation [170] as CWI[†] , because it brings no extra parameters and negligible extra costs. We use Batch Normalization [171] before CWI to normalize different fragments. Figure 3.4 illustrates the above procedure by taking $|\mathbb{I}| = 2$ as an example.

Discussion w.r.t. the sampling strategy in Eq. (3.12). This strategy aims to largely reduce the memory cost and training time to an acceptable amount by only back-propagating gradients of the sampled architectures instead of all architectures. Compared to sampling via a uniform distribution, the applied sampling method (sampling based on probability) could weaken the gradients difference caused by per-iteration sampling after multiple iterations.

Search for depth. We use parameters $\widehat{P^d} \in \mathbb{R}^L$ to indicate the distribution of the possible number of layers in a network with L convolutional layers. We utilize a similar strategy to sample the number of layers following Eq. (3.11) and allow $\widehat{P^d}$ to be differentiable as that of $\widehat{P^w}$, using the sampling distribution $\widetilde{P^d}_l$ for the depth l. We then calculate the final output feature of the pruned networks as an aggregation from all possible depths, which can be formulated as:

$$\boldsymbol{I}_{out} = \sum\nolimits_{l=1}^{L} \widetilde{P^d}_l \times \mathrm{CWI}(\hat{\boldsymbol{I}}^l, C_{out}), \tag{3.13}$$

where $\hat{\boldsymbol{I}}^l$ indicates the output feature map via Eq. (3.12) at the l-th layer. C_{out} indicates the maximum sampled channel among all $\hat{\boldsymbol{I}}^l$. The final output feature map $\boldsymbol{O}_{out}$ is fed into the last classification layer to make predictions. In this way, we can back-propagate gradients to both width parameters $\widehat{P^w}$ and depth parameters $\widehat{P^d}$.

[†]The formulation of the selected CWI: suppose $\boldsymbol{B} = \mathrm{CWI}(\boldsymbol{A}, C_{out})$, where $\boldsymbol{B} \in \mathbb{R}^{C_{out}HW}$ and $\boldsymbol{A} \in \mathbb{R}^{CHW}$; then $\boldsymbol{B}_{i,h,w} = \mathrm{mean}(\boldsymbol{A}_{s:e-1,h,w})$, where $s = \lfloor \frac{i \times C}{C_{out}} \rfloor$ and $e = \lceil \frac{(i+1) \times C}{C_{out}} \rceil$. We tried other forms of CWI, e.g., bilinear and trilinear interpolation. They obtain similar accuracy but are much slower than our choice.

Searching objectives. The final architecture α is derived by selecting the candidate with the maximum probability, learned by the architecture parameters $\mathcal{A}$, consisting of $\widehat{P^w}$ for each layers and $\widehat{P^d}$. Prevailing NAS methods [28, 33, 11, 72, 76] optimize α over network candidates with different typologies, while our TAS searches over candidates with the same typology structure as well as smaller width and depth. As a result, the validation loss in our search procedure includes not only the classification validation loss but also the penalty for the computation cost:

$$\mathcal{L}_{val} = -\log(\frac{\exp(\boldsymbol{z}_y)}{\sum_{j=1}^{|\boldsymbol{z}|}\exp(\boldsymbol{z}_j)}) + \lambda_{cost}\mathcal{L}_{cost}, \tag{3.14}$$

where $\boldsymbol{z}$ is a vector denoting the output logits from the pruned networks, y indicates the ground truth class of a corresponding input, and λ_{cost} is the weight of $\mathcal{L}_{cost}$. The cost loss encourages the computation cost of the network (e.g., FLOP) to converge to a target R so that the cost can be dynamically adjusted by setting different R. We used a piece-wise computation cost loss as:

$$\mathcal{L}_{cost} = \begin{cases} \log(\mathbb{E}_{cost}(\mathcal{A})) & F_{cost}(\mathcal{A}) > (1+t)\times R \\ 0 & (1-t)\times R < F_{cost}(\mathcal{A}) < (1+t)\times R\,, \\ -\log(\mathbb{E}_{cost}(\mathcal{A})) & F_{cost}(\mathcal{A}) < (1-t)\times R \end{cases} \tag{3.15}$$

where $\mathbb{E}_{cost}(\mathcal{A})$ computes the expectation of the computation cost, based on the architecture parameters $\mathcal{A}$. Specifically, it is the weighted sum of computation costs for all candidate networks, where the weight is the sampling probability. $F_{cost}(\mathcal{A})$ indicates the actual cost of the searched architecture, whose width and depth are derived from $\mathcal{A}$. $t \in [0, 1]$ denotes a toleration ratio, which slows down the speed of changing the searched architecture. Note that we use FLOP to evaluate the computation cost of a network, and it is readily to replace FLOP with other metric, such as latency [72].

Algorithm 2 The TAS Procedure

Input: split the training set into two disjoint sets: $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$

while not converge **do**
    Sample batch data $\mathbb{D}_t$ from $\mathcal{D}_{train}$
    Calculate the classification loss on $\mathbb{D}_t$ to update network weights
    Sample batch data $\mathbb{D}_v$ from $\mathcal{D}_{val}$
    Calculate the loss on $\mathbb{D}_v$ via Eq. (3.14) to update $\mathcal{A}$
end while
Derive the searched network from $\mathcal{A}$
Randomly initialize the searched network and optimize it by KD via Eq. (3.17) on the training set

We show the overall algorithm in Algorithm 2. During searching, we forward the network using Eq. (3.13) to make both weights and architecture parameters differentiable. We alternatively minimize the simple classification loss on the training set to optimize the pruned networks' weights and $\mathcal{L}_{val}$ on the validation set to optimize the architecture parameters $\mathcal{A}$. After searching, we pick up the number of channels with the maximum probability as width and the number of layers with the maximum probability as depth. The final searched network is constructed by the selected width and depth. This network will be optimized via KD, and we will introduced the details in Section 3.3.2.

Knowledge Transfer

Knowledge transfer is important to learn a robust pruned network, and we employ a simple KD algorithm [153] on a searched network architecture. This algorithm encourages the predictions $\boldsymbol{z}$ of the small network to match soft targets from the

unpruned network via the following objective:

$$\mathcal{L}_{\text{match}} = -\sum_{i=1}^{|\boldsymbol{z}|} \frac{\exp(\hat{\boldsymbol{z}}_i/T)}{\sum_{j=1}^{|\boldsymbol{z}|}\exp(\hat{\boldsymbol{z}}_j/T)} \log(\frac{\exp(\boldsymbol{z}_i/T)}{\sum_{j=1}^{|\boldsymbol{z}|}\exp(\boldsymbol{z}_j/T)}), \tag{3.16}$$

where T is a temperature, and $\hat{\boldsymbol{z}}$ indicates the logit output vector from the pre-trained unpruned network. Additionally, it uses a softmax with cross-entropy loss to encourage the small network to predict the true targets. The final objective of KD is as follows:

$$\mathcal{L}_{\text{KD}} = -\lambda \log(\frac{\exp(\boldsymbol{z}_y)}{\sum_{j=1}^{|\boldsymbol{z}|}\exp(\boldsymbol{z}_j)}) + (1-\lambda)\mathcal{L}_{\text{match}} \quad \text{s.t. } 0 \leq \lambda \leq 1, \tag{3.17}$$

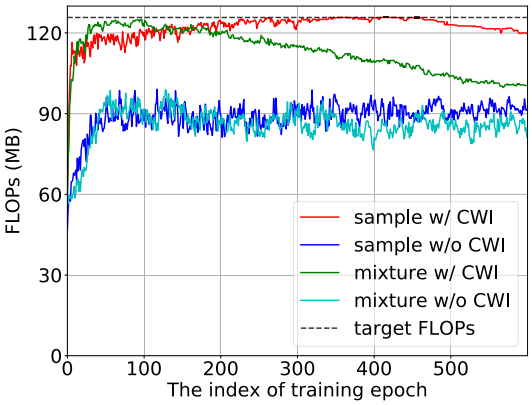
where y indicates the true target class of a corresponding input. λ is the weight of loss to balance the standard classification loss and soft matching loss. After we obtain the searched network, we first pre-train the unpruned network and then optimize the searched network by transferring from the unpruned network via Eq. (3.17).

3.3.3 Experimental Analysis

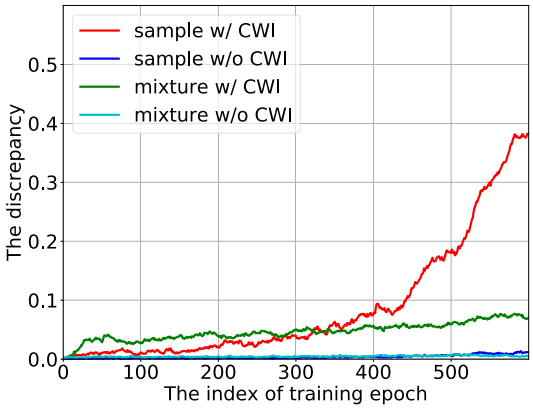
We introduce the experimental setup in Section 3.3.3. We evaluate different aspects of TAS in Section 3.3.3, such as hyperparameters, sampling strategies, different transferring methods, etc. Lastly, we compare TAS with other state-of-the-art pruning methods in Section 3.3.3.

Experimental Settings

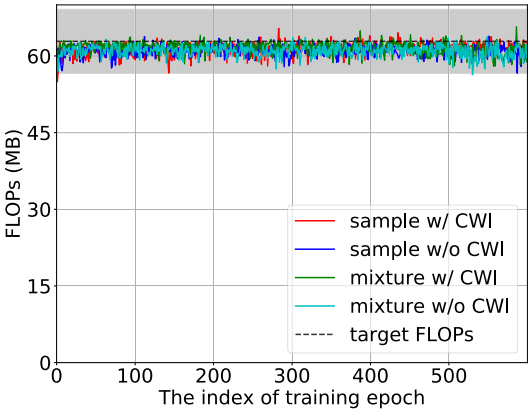
The search setting. We search the number of channels over {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0} of the original number in the unpruned network. We search the depth within each convolutional stage. We sample $|\mathbb{I}| = 2$ candidates in Eq. (3.12) to reduce the GPU memory cost during searching. We set R according to the FLOPs of the compared pruning algorithms and set λ_{cost} of 2. We optimize the weights via SGD and the architecture parameters via Adam. For the weights, we start the learning rate from 0.1 and reduce it by the cosine scheduler [40]. For the architecture



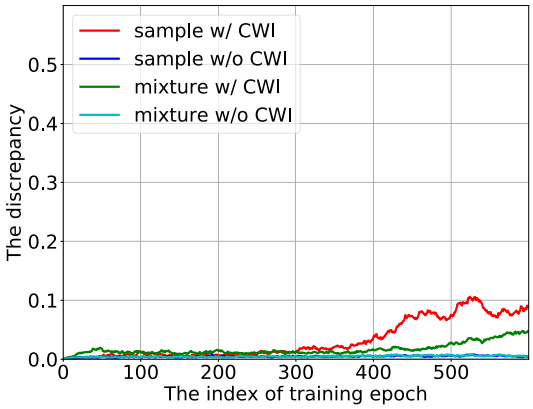
(a) The FLOPs of the searched network over epochs when we do not constrain the FLOPs ($\lambda_{cost} = 0$).



(b) The mean discrepancy over epochs when we do not constrain the FLOPs ($\lambda_{cost} = 0$).



(c) The FLOPs of the searched network over epochs when we constrain the FLOPs ($\lambda_{cost} = 2$).



(d) The mean discrepancy over epochs when we constrain the FLOPs ($\lambda_{cost} = 2$).

Figure 3.5 : The effect of different differentiable strategies.

parameters, we use the constant learning rate of 0.001 and a weight decay of 0.001. On both CIFAR-10 and CIFAR-100, we train the model for 600 epochs with the batch size of 256. On ImageNet, we train ResNets [20] for 120 epochs with the batch size of 256. The toleration ratio t is always set as 5%. The τ in Eq. (3.11) is linearly decayed from 10 to 0.1.

	FLOPs	Accuracy
Pre-defined	41.1 Million	68.18 %
Pre-defined w/ Init	41.1 Million	69.34 %
Pre-defined w/ KD	41.1 Million	71.40 %
Random Search	42.9 Million	68.57 %
Random Search w/ Init	42.9 Million	69.14 %
Random Search w/ KD	42.9 Million	71.71 %
TAS†	42.5 Million	68.95 %
TAS† w/ Init	42.5 Million	69.70 %
TAS† w/ KD (TAS)	42.5 Million	**72.41 %**

Table 3.5 : We compare the accuracy on CIFAR-100 when pruning about 40% FLOPs of ResNet-32.

Training. For CIFAR experiments, we use SGD with a momentum of 0.9 and a weight decay of 0.0005. We train each model by 300 epochs, start the learning rate at 0.1, and reduce it by the cosine scheduler [40]. We use the batch size of 256 and 2 GPUs. When using KD on CIFAR, we use λ of 0.9 and the temperature T of 4 following [154]. For ResNet models on ImageNet, we follow most hyperparameters as CIFAR, but use a weight decay of 0.0001. We use 4 GPUs to train the model by 120 epochs with the batch size of 256. When using KD on ImageNet, we set λ as 0.5 and T as 4 on ImageNet.

Case Studies

In this section, we evaluate different aspects of our proposed TAS. We also compare it with different searching algorithm and knowledge transfer method to demon-

#Selected Channels	Search Time	Memory	Train Time	FLOPs	Accuracy
$\|\mathbb{I}\|$=1	2.83 Hours	1.5GB	0.71 Hours	23.59 MB	89.85%
$\|\mathbb{I}\|$=2	3.83 Hours	2.4GB	0.84 Hours	38.95 MB	92.98%
$\|\mathbb{I}\|$=3	4.94 Hours	3.4GB	0.67 Hours	39.04 MB	92.63%
$\|\mathbb{I}\|$=5	7.18 Hours	5.1GB	0.60 Hours	37.08 MB	93.18%
$\|\mathbb{I}\|$=8	10.64 Hours	7.3GB	0.81 Hours	38.28 MB	92.65%

Table 3.6 : Results of different configurations when prune ResNet-32 on CIFAR-10.

strate the effectiveness of TAS.

The effect of different strategies to differentiate α. We apply our TAS on CIFAR-100 to prune ResNet-56. We try two different aggregation methods, i.e., using our proposed CWI to align feature maps or not. We also try two different kinds of aggregation weights, i.e., Gumbel-softmax sampling as Eq. (3.11) (denoted as "sample" in Figure 3.5) and vanilla-softmax as Eq. (3.10) (denoted as "mixture" in Figure 3.5). Therefore, there are four different strategies, i.e., with/without CWI combining with Gumbel-softmax/vanilla-softmax. Suppose we do not constrain the computational cost, then the architecture parameters should be optimized to find the maximum width and depth. This is because such network will have the maximum capacity and result in the best performance on CIFAR-100. We try all four strategies with and without using the constraint of computational cost. We show the results in Figure 3.5c and Figure 3.5a. When we do not constrain the FLOPs, our TAS can successfully find the best architecture should have a maximum width and depth. However, other three strategies failed. When we use the FLOP constraint, we can successfully constrain the computational cost in the target range. We also investigate discrepancy between the highest probability and the second highest prob-

ability in Figure 3.5d and Figure 3.5b. Theoretically, a higher discrepancy indicates that the model is more confident to select a certain width, while a lower discrepancy means that the model is confused and does not know which candidate to select. As shown in Figure 3.5d, with the training procedure going, our TAS becomes more confident to select the suitable width. In contrast, strategies without CWI can not optimize the architecture parameters; and "mixture with CWI" shows a worse discrepancy than ours.

Comparison w.r.t. structure generated by different methods in Table 3.5. "Pre-defined" means pruning a fixed ratio at each layer [149]. "Random Search" indicates an NAS baseline used in [28]. "TAS†" is our proposed differentiable searching algorithm. We make two observations: (1) searching can find a better structure using different knowledge transfer methods; (2) our TAS is superior to the NAS random baseline.

Comparison w.r.t. different knowledge transfer methods in Table 3.5. The first line in each block does not use any knowledge transfer method. "w/ Init" indicates using pre-trained unpruned network as initialization. "w/ KD" indicates using KD. From Table 3.5, knowledge transfer methods can consistently improve the accuracy of pruned network, even if a simple method is applied (Init). Besides, KD is robust and improves the pruned network by more than 2% accuracy on CIFAR-100.

Searching width vs. searching depth. We try (1) only searching depth ("TAS (D)"), (2) only searching width ("TAS (W)"), and (3) searching both depth and width ("TAS") in Table 3.7. Results of only searching depth are worse than results of only searching width. If we jointly search for both depth and width, we can achieve better accuracy with similar FLOP than both searching depth and searching width only.

The effect of selecting different numbers of architecture samples $\mathbb{I}$ in

Model	Method	CIFAR-10			CIFAR-100		
		Pruned Model Accuracy	Accuracy Drop	FLOPs (pruning ratio)	Pruned Model Accuracy	Accuracy Drop	FLOPs (pruning ratio)
ResNet-20	LCCL [14]	91.68%	1.06%	2.61E7 (36.0%)	64.66%	2.87%	2.73E7 (33.1%)
	SFP [148]	90.83%	1.37%	2.43E7 (42.2%)	64.37%	3.25%	2.43E7 (42.2%)
	FPGM [166]	91.09%	1.11%	2.43E7 (42.2%)	66.86%	0.76%	2.43E7 (42.2%)
	TAS (D)	90.97%	1.91%	2.19E7 (46.2%)	64.81%	3.88%	2.19E7 (46.2%)
	TAS (W)	92.31%	0.57%	1.99E7 (51.3%)	68.08%	0.61%	1.92E7 (52.9%)
	TAS	92.88%	0.00%	2.24E7 (45.0%)	68.90%	-0.21%	2.24E7 (45.0%)
ResNet-32	LCCL [14]	90.74%	1.59%	4.76E7 (31.2%)	67.39%	2.69%	4.32E7 (37.5%)
	SFP [148]	92.08%	0.55%	4.03E7 (41.5%)	68.37%	1.40%	4.03E7 (41.5%)
	FPGM [166]	92.31%	0.32%	4.03E7 (41.5%)	68.52%	1.25%	4.03E7 (41.5%)
	TAS (D)	91.48%	2.41%	4.08E7 (41.0%)	66.94%	3.66%	4.08E7 (41.0%)
	TAS (W)	92.92%	0.96%	3.78E7 (45.4%)	71.74%	-1.12%	3.80E7 (45.0%)
	TAS	93.16%	0.73%	3.50E7 (49.4%)	72.41%	-1.80%	4.25E7 (38.5%)
ResNet-56	PFEC [149]	93.06%	-0.02%	9.09E7 (27.6%)	–	–	–
	LCCL [14]	92.81%	1.54%	7.81E7 (37.9%)	68.37%	2.96%	7.63E7 (39.3%)
	AMC [151]	91.90%	0.90%	6.29E7 (50.0%)	–	–	–
	SFP [148]	93.35%	0.56%	5.94E7 (52.6%)	68.79%	2.61%	5.94E7 (52.6%)
	FPGM [166]	93.49%	0.42%	5.94E7 (52.6%)	69.66%	1.75%	5.94E7 (52.6%)
	TAS	93.69%	0.77%	5.95E7 (52.7%)	72.25%	0.93%	6.12E7 (51.3%)
ResNet-110	LCCL[14]	93.44%	0.19%	1.66E8 (34.2%)	70.78%	2.01%	1.73E8 (31.3%)
	PFEC [149]	93.30%	0.20%	1.55E8 (38.6%)	–	–	–
	SFP [148]	92.97%	0.70%	1.21E8 (52.3%)	71.28%	2.86%	1.21E8 (52.3%)
	FPGM [166]	93.85%	-0.17%	1.21E8 (52.3%)	72.55%	1.59%	1.21E8 (52.3%)
	TAS	94.33%	0.64%	1.19E8 (53.0%)	73.16%	1.90%	1.20E8 (52.6%)
ResNet-164	LCCL[14]	94.09%	0.45%	1.79E8 (27.40%)	75.26%	0.41%	1.95E8 (21.3%)
	TAS	94.00%	1.47%	1.78E8 (28.10%)	77.76%	0.53%	1.71E8 (30.9%)

Table 3.7 : Comparison of different pruning algorithms for ResNet.

Eq. (3.12). We compare different numbers of selected channels in Table 3.6 and did experiments on a single NVIDIA Tesla V100. The searching time and the GPU memory usage will increase linearly to $|\mathbb{I}|$. When $|\mathbb{I}| = 1$, since the re-normalized

probability in Eq. (3.12) becomes a constant scalar of 1, the gradients of parameters α will become 0 and the searching failed. When $|\mathbb{I}| > 1$, the performance for different $|\mathbb{I}|$ is similar.

The speedup gain. As shown in Table 3.6, TAS can finish the searching procedure of ResNet-32 in about 3.8 hours on a single V100 GPU . If we use ES or random searching methods, we need to train network with many different candidate configurations one by one and then evaluate them to find a best. In this way, much more computational costs compared to our TAS are required. A possible solution to accelerate ES or random searching methods is to share parameters of networks with different configurations [29, 172].

Compared to the state-of-the-art

Results on CIFAR in Table 3.7. We prune different ResNets on both CIFAR-10 and CIFAR-100. Most previous algorithms perform poorly on CIFAR-100, while our TAS consistently outperforms then by more than 2% accuracy in most cases. On CIFAR-10, our TAS outperforms the state-of-the-art algorithms on ResNet-20,32,56,110. For example, TAS obtains 72.25% accuracy by pruning ResNet-56 on CIFAR-100, which is higher than 69.66% of FPGM [166]. For pruning ResNet-32 on CIFAR-100, we obtain greater accuracy and less FLOP than the unpruned network. We obtain a slightly worse performance than LCCL [14] on ResNet-164. It because there are $8^{163} \times 18^3$ candidate network structures to searching for pruning ResNet-164. It is challenging to search over such huge search space, and the very deep network has the over-fitting problem on CIFAR-10 [20].

Results on ImageNet in Table 3.8. We prune ResNet-18 and ResNet-50 on ImageNet. For ResNet-18, it takes about 59 hours to search for the pruned network on 4 NVIDIA Tesla V100 GPUs. The training time of unpruned ResNet-18 costs about 24 hours, and thus the searching time is acceptable. With more machines

Model	Method	ImageNet Top-1		ImageNet Top-5		FLOPs	Prune Ratio
		Prune Acc	Acc Drop	Prune Acc	Acc Drop		
ResNet-18	LCCL [14]	66.33%	3.65%	86.94%	2.29%	1.19E9	34.6%
	SFP [148]	67.10%	3.18%	87.78%	1.85%	1.06E9	41.8%
	FPGM [166]	68.41%	1.87%	88.48%	1.15%	1.06E9	41.8%
	TAS	69.15%	1.50%	89.19%	0.68%	1.21E9	33.3%
ResNet-50	SFP [148]	74.61%	1.54%	92.06%	0.81%	2.38E9	41.8%
	CP [147]	-	-	90.80%	1.40%	2.04E9	50.0%
	AutoSlim [172]	76.00%	-	-	-	3.00E9	26.6%
	FPGM [166]	75.50%	0.65%	92.63%	0.21%	2.36E9	42.2%
	TAS	76.20%	1.26%	93.07%	0.48%	2.31E9	43.5%

Table 3.8 : Comparison of different pruning algorithms.

and optimized implementation, we can finish TAS with less time cost. We show competitive results compared to other state-of-the-art pruning algorithms. For example, TAS prunes ResNet-50 by 43.5% FLOPs, and the pruned network achieves 76.20% accuracy, which is higher than FPGM by 0.7. Similar improvements can be found when pruning ResNet-18. Note that we directly apply the hyperparameters on CIFAR-10 to prune models on ImageNet, and thus TAS can potentially achieve a better result by carefully tuning parameters on ImageNet.

Our proposed TAS is a preliminary work for the new network pruning pipeline. This pipeline can be improved by designing more effective searching algorithm and knowledge transfer method. We hope that future work to explore these two components will yield powerful compact networks.

3.4 Conclusion

In this chapter, firstly, we proposed a Gradient-based searching approach using Differentiable Architecture Sampling (GDAS) to achieve the second objective in this thesis (Obj-2, "Design efficient AutoDL algorithm for searching for neural architecture topology.") Later, extensive experiments on the CIFAR-10, CIFAR-100, ImageNet, PTB, and WT2 datasets demonstrate the effectiveness and efficiency of our proposed GDAS. These comprehensive evaluations help us to achieve the third objective (Obj-3, "Systemically evaluate the proposed efficient algorithm on computer vision and natural language processing applications."). Lastly, we generalized the idea of differentiable architecture sampling to search for the architecture size, and proposed a new NAS algorithm – Transformable Architecture Search (TAS). To evaluate the proposed TAS algorithm, we systematically analyze it on the CIFAR and ImageNet datasets. Such a generalization achieved the fourth objective (Obj-4, "Generalize the proposed efficient algorithm to handle both neural architecture topology and neural architecture size.").

Chapter 4

AutoHAS: Differentiable Hyperparameter and Architecture Search

4.1 Introduction

In the last chapter Chapter 3, we took the first step to solve the efficient searching problem for neural architecture. In this chapter, we target a more challenging problem, i.e., studying how to jointly search for both hyperparameters and architectures, aiming to answer the second question described in Chapter 1 – (Q-2 "How could we design a unified and efficient AutoDL framework?"). We first mathematically re-formulate the joint searching problem as a bi-level optimization problem, and then propose a new unified framework for it. As a result, the objective Obj-5 identified in Chapter 1 will be achieved.

NAS has brought significant improvements in many applications, such as machine perception [173, 97, 174, 175, 176], language modeling [28, 11], and model compression [151, 145]. Most NAS works apply the *same* hyperparameters while searching for network architectures. For example, all models in [33, 177, 98] are trained with the same optimizer, learning rate, and weight decay. As a result, the relative ranking of models in the search space is *only* determined by their architectures. However, we observe that different models favor different hyperparameters. Table 4.1 shows the performance of two randomly sampled models with different hyperparameters: under hyperparameter HP_1, model_1 outperforms model_2, but model_2 is better under HP_2. These results suggest using fixed hyperparameters in NAS would lead to sub-optimal results.

	Model_1	rank	Model_2
HP_1 (LR=5.5,L2=1.5e-4)	**56.9%**	>	55.6%
HP_2 (LR=1.1,L2=8.4e-4)	54.7%	<	**56.2%**

Table 4.1 : ImageNet accuracy of two models randomly sampled from search space based on MobileNet-V2 [15]. Model_1 favors HP_1 while Model_2 favors HP_2.

A natural question is: could we extend NAS to a broader scope for joint Hyperparameter and Architecture Search (HAS)? In HAS, each model can potentially be coupled with its own best hyperparameters, thus achieving better performance than existing NAS with *fixed* hyperparameters. However, jointly searching for architectures and hyperparameters is challenging. The first challenge is how to deal with both categorical and continuous values in the joint HAS search space. While architecture choices are mostly categorical values (e.g., convolution kernel size), hyperparameters choices can be both categorical (e.g., the type of optimizer) and continuous values (e.g., weight decay). There is not yet a good solution to tackle this challenge: previous NAS methods *only* focus on categorical search spaces, while hyperparameter optimization methods *only* focus on continuous search spaces. They thus cannot be directly applied to such a mixture of categorical and continuous search space. Secondly, another critical challenge is how to efficiently search over the larger joint HAS search space as it combines both architecture and hyperparameter choices.

In this chapter, we propose *AutoHAS*, a differentiable HAS algorithm. It is, to the best of our knowledge, the first algorithm that can efficiently handle the large joint HAS search space. To address the mixture of categorical and continuous search spaces, we first discretize the continuous hyperparameters into a linear combination of multiple categorical basis, then we can unify them during search. As explained below, we will use a differentiable method to search over the combination, i.e., archi-

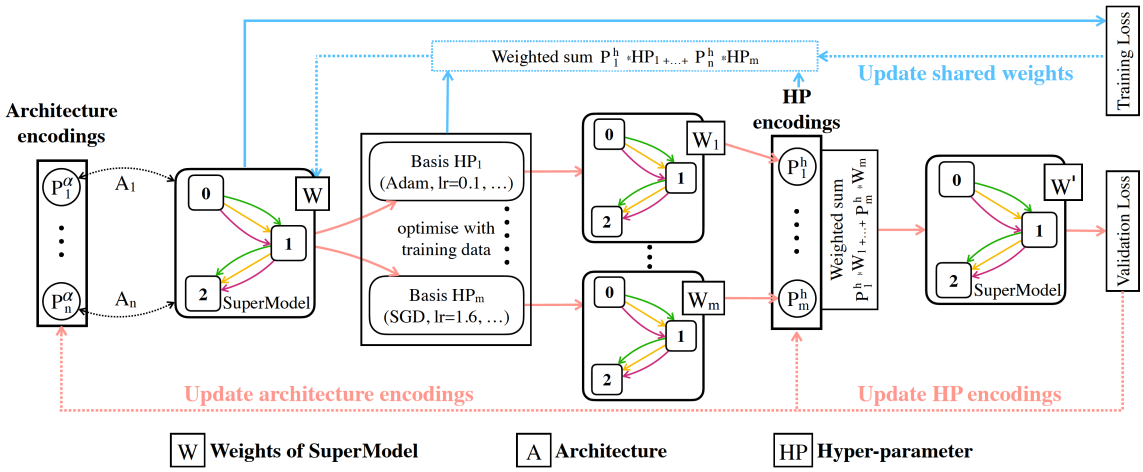


Figure 4.1 : The AutoHAS framework.

tecture and HP encodings in Figure 4.1. These encodings represent the probability distribution over all candidates in the respective space. They can be used to find the best architecture together with its associated hyperparameters.

To efficiently navigate the much larger search space, we further introduce a novel weight sharing technique for *AutoHAS*. Weight sharing has been widely used in previous NAS approaches [29, 28] to reduce the search cost. The main idea is to train a SuperModel, where each candidate in the architecture space is its sub-model. Using a SuperModel can avoid training millions of candidates from scratch [28, 11, 72, 29]. Motivated by the weight sharing in NAS, *AutoHAS* extends its scope from architecture search to both architecture and hyperparameter search. We not only share the weights of the SuperModel with each architecture but also share this SuperModel across hyperparameters. At each search step, *AutoHAS* optimizes the shared SuperModel by a combination of the basis of HAS space, and the shared SuperModel serves as a good initialization for all hyperparameters at the next step of search (see Figure 4.1 and Section 4.2).

In this chapter, we focus on architecture, learning rate, and L2 penalty weight optimization, but it should be straightforward to apply *AutoHAS* to other hyper-

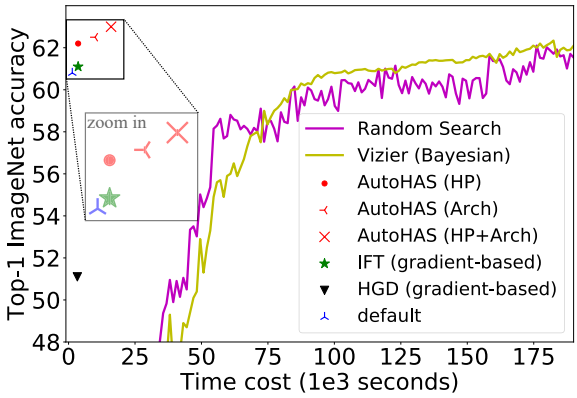


Figure 4.2 : AutoHAS achieves higher accuracy with 10× less search cost than other AutoML methods.

parameters. A summary of our results is in Figure 4.2, which shows that *AutoHAS* outperforms many AutoML methods regarding both accuracy and efficiency (more details in Section 4.3.3). In Section 4.3, we show that it improves a number of computer vision and natural language processing models, i.e., MobileNet-V2 [15], ResNet [20], EfficientNet [177], and BERT fine-tuning [178].

4.2 Methodology

In this section, we will elaborate on the core ideas of *AutoHAS*. We provide some background of HAS in Section 4.2.1, and then introduce how to unify the joint categorical and continuous search space in Section 4.2.2. Afterwards, we describe an efficient *AutoHAS* searching algorithm in Section 4.2.3 and show how to derive the final hyperparameters and architecture in Section 4.2.4.

4.2.1 Preliminaries

HAS aims to find architecture α and hyperparameters h that achieve high performance on the validation set. HAS can be formulated as a bi-level optimization

problem:

$$\min_{\alpha,h} \mathcal{L}(\alpha, h, \omega_\alpha^*, \mathbb{D}_{val}) \quad \text{s.t. } \omega_\alpha^* = f_h(\alpha, \omega_\alpha^0, \mathbb{D}_{train}), \tag{4.1}$$

where $\mathcal{L}$ indicates the objective function (e.g., cross-entropy loss) and ω_α^0 indicates the initial weights of the architecture α. $\mathbb{D}_{train}$ and $\mathbb{D}_{val}$ denote the training data and the validation data, respectively. f_h represents the algorithm with hyperparameters h to obtain the optimal weights ω_α^*, such as using SGD to minimize the training loss. In that case, $\omega_\alpha^* = f_h(\alpha, \omega_\alpha^0, \mathbb{D}_{train}) = \arg\min_{\omega_\alpha} \mathcal{L}(\alpha, h, \omega_\alpha^0, \mathbb{D}_{train})$. We can also use HyperNetwork [56] to generate weights ω_α^*.

HAS generalizes both NAS and HPO by introducing a broader search space. On one-hand, NAS is a special case of HAS, where the inner optimization $f_h(\alpha, \omega_\alpha^0, \mathbb{D}_{train})$ uses fixed α and h to optimize $\min_\omega \mathcal{L}(\alpha, h, \omega, \mathbb{D}_{train})$. On the other, HPO is a special case of HAS, where α is fixed in Eq. (4.1).

4.2.2 Representation of the HAS Search Space in *AutoHAS*

The search space of HAS in *AutoHAS* is a Cartesian product of the architecture and hyperparameter candidates. To search over this mixed search space, we need a unified representation of different searchable components, i.e., architectures, learning rates, optimizers, etc.

Architectures Search Space. We use the simplest case as an example. First of all, let the set of predefined candidate operations (e.g., 3x3 convolution, pooling, etc.) be $\mathcal{O} = \{O_1, O_2, ..., O_n\}$, where the cardinality of $\mathcal{O}$ is n. Suppose an architecture is constructed by stacking multiple layers, each layer takes a tensor F as input and output $\pi(F)$, which serves as the next layer's input. $\pi \in \mathcal{O}$ denotes the operation at a layer and might be different at different layers. Then a candidate architecture α is essentially the sequence for all layers $\{\pi\}$. Further, a layer can be represented

as a linear combination of the operations in $\mathcal{O}$ as follows:

$$\pi(F) = \sum_{i=1}^{n} P_i^{\alpha} \, O_i(F) \qquad \text{s.t. } P_i^{\alpha} \in \{0,1\}, \sum_{i=1}^{n} P_i^{\alpha} = 1, \tag{4.2}$$

where P_i^{α} (the i-th element of the vector P^{α}) is the coefficient of operation O_i for a layer. We call the set of all coefficients $\mathcal{A} = \{P^{\alpha} \text{ for all layers}\}$ the *architecture encoding*, which can then represent the search space of the architecture.

Hyperparameter Search Space. Now we can define the hyperparameter search space in a similar way. The major difference is that we have to consider both categorical and continuous cases:

$$h = \sum_{i=1}^{m} P_i^{h} \, \mathcal{B}_i \qquad \text{s.t. } \sum_{i=1}^{m} P_i^{h} = 1, \; P_i^{h} \in \begin{cases} [0,1], & \text{if continuous} \\ \{0,1\}, & \text{if categorical} \end{cases} \tag{4.3}$$

where $\mathcal{B}$ is a predefined set of hyperparameter basis with the cardinality of m and $\mathcal{B}_i$ is the i-th basis in $\mathcal{B}$. P_i^{h} (the i-th element of the vector P^{h}) is the coefficient of hyperparameter basis $\mathcal{B}_i$. If we have a continuous hyperparameter, we have to discretize it into a linear combination of basis and unify both categorical and continuous. For example, for weight decay, $\mathcal{B}$ could be {1e-1, 1e-2, 1e-3}, and therefore, all possible weight decay values can be represented as a linear combination over $\mathcal{B}$. For categorical hyperparameters, taking the optimizer as an example, $\mathcal{B}$ could be {Adam, SGD, RMSProp}. In this case, a constraint on P_i^{h} is applied: $P_i^{h} \in \{0,1\}$ as in Eq. (4.3). When there are multiple different types of hyperparameters, each of them will have their own P^{h}. The hyperparameter basis becomes the Cartesian product of their own basis and the coefficient is the product of the corresponding P_i^{h}. We name the set of all coefficients $\mathcal{H} = \{P^{h} \text{ for all types of hyperparameter}\}$ the *hyperparameter encoding*, which can then represent the search space of hyperparameters.

4.2.3 Automated Hyperparameter and Architecture Search

Since each candidate in the HAS search space can be represented by a pair of $\mathcal{H}$ and $\mathcal{A}$, the searching problem is converted to optimizing the encoding $\mathcal{H}$ and $\mathcal{A}$. However, it is computationally prohibitive to compute the exact gradient of $\mathcal{L}(\alpha, h, \omega_\alpha^*, \mathbb{D}_{val})$ in Eq. (4.1) w.r.t. $\mathcal{H}$ and $\mathcal{A}$. Alternatively, we propose a simple approximation strategy with weight sharing to accelerate this procedure.

First of all, we leverage a SuperModel to share weights among all candidate architectures in the architecture space, where each candidate is a sub-model in this SuperModel [29, 28]. The weights of the SuperModel $\mathcal{W}$ is the union of weights of all basis operations in each layer. The weights ω_α of an architecture α can thus be represented by $\mathcal{W}_\alpha$, a subset of $\mathcal{W}$. Computing the exact gradients of $\mathcal{L}$ w.r.t. $\mathcal{H}$ and $\mathcal{A}$ requires back-propagating through the initial network state $\mathcal{W}_\alpha^0$, which is too expensive. Inspired by [28, 29], we approximate it using the current SuperModel weight $\mathcal{W}$ as follows:

$$\nabla_{\mathcal{A},\mathcal{H}} \mathcal{L}(\alpha, h, \omega_\alpha^*, \mathbb{D}_{val}) \approx \nabla_{\mathcal{A},\mathcal{H}} \mathcal{L}(\alpha, h, f_h(\alpha, \mathcal{W}_\alpha, \mathbb{D}_{train}), \mathbb{D}_{val}), \tag{4.4}$$

Ideally, we should back-propagate $\mathcal{L}$ through f_h to modify the encoding $\mathcal{H}$. However, f_h might be a complex optimization algorithm and not allow back-propagation. To solve this problem, we regard f as a black-box function and reformulate f_h as follows:

$$f_h(\alpha, \mathcal{W}_\alpha, \mathbb{D}_{train}) \approx \sum_{i=1}^{m} P_i^h \; f_{\mathcal{B}_i}(\alpha, \mathcal{W}_\alpha, \mathbb{D}_{train}), \tag{4.5}$$

In this way, $f_h(\alpha, \mathcal{W}_\alpha, \mathbb{D}_{train})$ is calculated as a weighted sum of P_i^h and generated weights from $f_{\mathcal{B}_i}$.

In practice, it is not easy to directly optimize the encodings $\mathcal{A}$ and $\mathcal{H}$, because they naturally have some constraints associated with them, such as Eq. (4.3). Inspired by the continuous relaxation [28, 11], we instead use another set of relaxed variables $\widetilde{\mathcal{A}} = \{\widetilde{P}^\alpha \text{ for all layers}\}$ and $\widetilde{\mathcal{H}} = \{\widetilde{P}^h \text{ for all types of hyperparameters}\}$

to calculate $\mathcal{A}$ and $\mathcal{H}$. $\widetilde{P}^{\alpha}$ and $\widetilde{P}^{h}$ have the same dimension as P^{α} and P^{h}. The calculation procedure encapsulates the constraints of P^{α} and P^{h} in Eq. (4.2) and Eq. (4.3) as follows:

$$P^{h} = one\_hot(\arg\max_{j} \bar{P}^{h}_{j}), \tag{4.6}$$

$$\bar{P}^{h}_{i} = \frac{\exp((\widehat{P}^{h}_{i} + \boldsymbol{o}_{i})/\tau)}{\sum_{k}\exp((\widehat{P}^{h}_{k} + \boldsymbol{o}_{k})/\tau)}, \quad \text{where} \quad \widehat{P}^{h}_{i} = \frac{\exp(\widetilde{P}^{h}_{i})}{\sum_{k}\exp(\widetilde{P}^{h}_{k})}, \tag{4.7}$$

$$\boldsymbol{o}_{k} = -\log(-\log(u)), \quad \text{where} \quad u \sim \text{Uniform}[0,1], \tag{4.8}$$

where P^{h} is computed by applying the Gumbel-Softmax function [65, 128] on $\widetilde{P}^{h}$. τ is a temperature value and $\boldsymbol{o}_{k}$ are i.i.d samples drawn from Gumbel (0,1). The Gumbel-Softmax in Eq. (4.7) incorporates the stochastic procedure during search. It can help explore more candidates in the HAS search space and avoid over-fitting to some sub-optimal architecture and hyperparameters. We use the same procedure as Eq. (4.6)$\sim$(4.8) to define $\bar{P}^{\alpha}$ and $\widehat{P}^{\alpha}$ for architecture encodings. Ideally, the encodings should be optimized with Eq. (4.6) by back-propagation, but unfortunately one-hot encodings P^{h} and P^{α} are not differentiable. To address this issue, we follow [11, 65, 128] to relax the one-hot encodings: in the forward pass, we use one-hot encodings P^{h} to compute validation loss, but in the backward pass, we apply relaxation on P^{h} and substitute $\frac{\partial P^{h}}{\partial \widetilde{P}^{h}}$ by $\frac{\partial \bar{P}^{h}}{\partial \widetilde{P}^{h}}$ during back-propagation.

We describe our *AutoHAS* algorithm in Algorithm 3. During search, we jointly optimize $\mathcal{W}$ and $(\widetilde{\mathcal{A}}, \widetilde{\mathcal{H}})$ in an iterative way. The $\mathcal{W}$ is updated as follows:

$$\mathcal{W}_{\alpha} \leftarrow f_{h}(\alpha, \mathcal{W}_{\alpha}, \mathbb{D}_{train}), \tag{4.9}$$

where f_{h} is a training algorithm: in our experiments, it is implemented as minimizing the training loss with respect to hyperparameter h by one step. Notably, in Eq. (4.9), h is computed by $\widetilde{\mathcal{H}}$ and α is computed by $\widetilde{\mathcal{A}}$.

4.2.4 Deriving Hyperparameters and Architecture

Algorithm 3 The AutoHAS Algorithm

Input: Randomly initialize $\mathcal{W}$

Input: Randomly initialize $(\widetilde{\mathcal{A}}, \widetilde{\mathcal{H}})$

Input: Split the available data into two disjoint sets: $\mathbb{D}_{train}$ and $\mathbb{D}_{val}$

1: **while** not converged **do**

2: Update weights $\mathcal{W}$ via Eq. (4.9)

3: Optimize $(\widetilde{\mathcal{A}}, \widetilde{\mathcal{H}})$ via Eq. (4.4)$\sim$(4.8)

4: **end while**

5: Derive the final architecture from $\widetilde{\mathcal{A}}$ and hyperparameters from $\widetilde{\mathcal{H}}$

After obtaining the optimized encoding of architecture $\widetilde{\mathcal{A}} = \{\widetilde{P}^{\alpha}\}$ and hyperparameters $\widetilde{\mathcal{H}} = \{\widetilde{P}^{h}\}$ following Section 4.2.3, we use them to derive the final architecture and hyperparameters. For hyperparameters, we apply different strategies to the continuous and categorical values:

$$P^h = \begin{cases} \widehat{P}^h & \text{if continuous} \\ \text{one\_hot}(\arg\max_i \widehat{P}^h_i) & \text{if categorical} \end{cases}, \tag{4.10}$$

For architectures, since all values are categorical, we apply the same strategy in Eq. (4.10) for categorical values.

Notably, unlike other fixed hyperparameters, the learning rate can have different values at each training step, so it is a list of continuous values instead of a single scalar. To deal with this special case, we use Eq. (4.10) to derive the continuous learning rate value at each searching step, such that we can obtain a list of learning rate values corresponding to each specific step.

After we derive the final architecture and hyperparameters as in Algorithm 3, we will use the searched hyperparameters to re-train the searched architecture.

4.3 Experiments

4.3.1 Experimental Settings

Datasets. We demonstrate the effectiveness of our *AutoHAS* on five vision datasets, i.e., ImageNet [179], Birdsnap [180], CIFAR-10, CIFAR-100 [132], and Cars [181], and a NLP dataset, i.e., SQuAD 1.1 [182].

Searching settings. We call the hyperparameters that control the behavior of *AutoHAS* as meta hyperparameters. For the meta hyperparameters, we set $\tau = 10$ in Gumbel-Softmax and employ Adam optimizer with a fixed learning rate 0.002. Notably, *we use the same meta hyperparameters for all search experiments.* The number of searching epochs and batch size are set to be the same as in the training settings of baseline models, i.e., they can be different for different baseline models. When searching for MBConv-based models [60, 15], we search for the kernel size from {3, 5, 7} and the expansion ratio from {3, 6}. For vision tasks, the hyperparameter basis for the continuous value is the product of the default value and multipliers {0.1, 0.25, 0.5, 0.75, 1.0, 2.5, 5.0, 7.5, 10.0}. For the NLP task, we use smaller multipliers {0.01, 0.05, 0.1, 0.5, 1.0, 1.2, 1.5} since they are for fine-tuning on top of pretrained models. If a model has a default learning rate schedule, we create a range of values around the default learning rate at each step and use *AutoHAS* to find the best learning rate at each step.

Training settings. On vision datasets, we use six models, i.e., three variants of MobileNet-V2 (MNet2), EfficientNet-A0 (ENet-A0), ResNet-18, and ResNet-50. We use the batch size of 4096 for ImageNet and 1024 for other vision datasets. We use the same data augmentation as shown in [20]. On the NLP dataset SQuAD, we fine-tune the pretrained $\text{BERT}_{\text{LARGE}}$ model and follow the setting of [178]. The number of training epochs is different for different datasets, and we will explain their details later. For learning rate and weight decay, we use the values found by

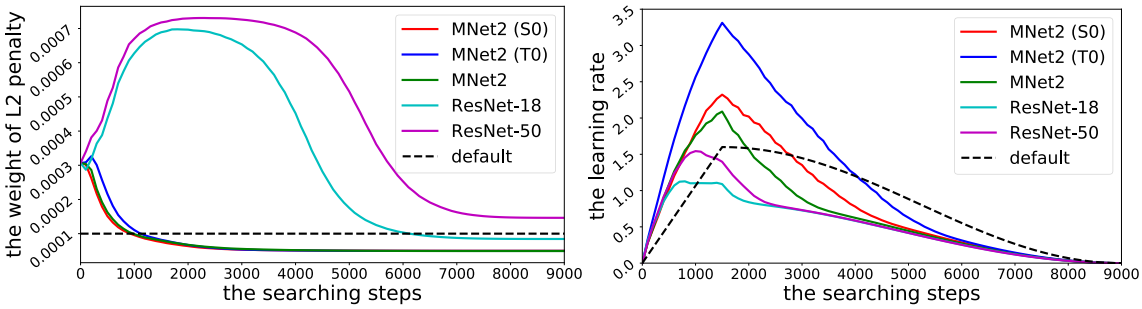


Figure 4.3 : AutoHAS found different learning rate and weight of L2 penalty for different models.

AutoHAS.

4.3.2 Ablation Studies

Searching Strategy	Deriving Strategy	MNet2 (S0)	MNet
Softmax	Eq. (4.6)	44.0%	63.5%
Gumbel-Softmax (soft)	Eq. (4.6)	45.5%	65.2%
Gumbel-Softmax (hard)	Eq. (4.6)	45.9%	66.4%
Softmax	Eq. (4.10)	40.8%	61.4%
Gumbel-Softmax (soft)	Eq. (4.10)	41.5%	67.0%
Gumbel-Softmax (hard)	Eq. (4.10)	46.3%	67.5%

Table 4.2 : We analyze different strategies used in *AutoHAS*.

We did a series of experiments to study the effect of (I) different searching strategies; (II) different deriving strategies; (III) *AutoHAS*-searched vs. manually tuned hyperparameters.

The effect of searching strategies. One of the key questions in searching is how to relax and optimize the architecture and hyperparameter encoding. Our

AutoHAS leverages Gumbel-Softmax in Eq. (4.7) to stochastically explore different hyperparameter and architecture basis. We evaluate two different variants in Table 4.2. "Softmax" does not add the Gumbel distributed noise and performs poorly compared to using Gumbel-Softmax. This strategy has an over-fitting problem, which is also found in NAS [11, 81, 6, 176]. "GS (soft)" does not use the one-hot vector in Eq. (4.7) and thus it will explore too many hyperparameters during searching. As a result, its optimization might become difficult and the found are worse than *AutoHAS*.

The effect of deriving strategies. We evaluate two kinds of strategies to derive the final hyperparameters and architectures. The vanilla strategy is to follow previous NAS methods: selecting the basis hyperparameters with the maximal probability. However, it does not work well for the continuous choices. As shown in Table 4.2, our proposed strategy "GS (hard) + Eq. (4.10)" can improve the accuracy by 1.1% compared to the vanilla strategy "GS (hard) + Eq. (4.6)".

Searched hyperparameters vs. manually tuned hyperparameters. We show the searched and manually tuned hyperparameters in Figure 4.3. For the weight of L2 penalty, it is interesting that *AutoHAS* indicates using large penalty for the large models (ResNet) at the beginning and decay it to a smaller value at the end of searching. For manual tuning, you need to tune every model one by one to obtain their optimal hyperparameters. In contrast, *AutoHAS* only requires to tune *two* meta hyperparameters, in which it can successfully find good hyperparameters for tens of models. Besides, some hyperparameters, such as learning rate, are dynamically changed for every training step. It is hard for human to tune its per-step value, while *AutoHAS* can deal with such hyperparameters.

4.3.3 AutoHAS for Vision Datasets

ImageNet: We first apply AutoHAS to ImageNet and compare the performance with previous AutoML algorithms. We choose the hyperparameters used for ResNet [20] as our default hyperparameters: warm-up the learning rate at the first 5 epochs from 0 to $0.1 \times \frac{batch\ size}{256}$ and decay it to 0 via cosine annealing schedule [183]; use the weight of L2 penalty as 1e-4. Since these hyperparameters have been heavily tuned by human experts, there is limited headroom to improve. Therefore, we study how to train a model to achieve a good performance in shorter time, i.e., 30 epochs.

Table 4.3 and Table 4.4 shows the performance comparison. There are some interesting observations: (I) *AutoHAS* is applicable to searching for almost all kinds of hyperparameters and architectures, while previous hyper-gradient based methods [48, 92] can only be applied to some hyperparameters. (II) *AutoHAS* shows improvements in seven different representative models, including both light-weight and heavy models. (III) The found hyperparameters by *AutoHAS* outperform the (default) manually tuned hyperparameters. (IV) The found hyperparameters by *AutoHAS* outperform that found by other AutoML algorithms. (V) Searching over the large joint HAS search space can obtain better results compared to searching for hyperparameters only. (VI) Gradient-based AutoML algorithms are more efficient than black-box optimization methods, such as random search and vizier.

Smaller datastes: To analyze the effect of architecture and hyperparameters, we compare *AutoHAS* with two variants: searching for architecture only, i.e., GDAS (Arch), and searching for hyperparameters only, i.e., *AutoHAS* (HP). The results on four datasets are shown in Table 4.5. On Birdsnap and Cars, *AutoHAS* significantly outperforms GDAS (Arch) and *AutoHAS* (HP). On CIFAR-100, the accuracy of *AutoHAS* is similar to GDAS (Arch) and *AutoHAS* (HP), while all of them outperform the default. On CIFAR-10, the accuracy of auto-tuned architecture or

Type	Model	Searching Methods					
		default HP	RS [24]	Vizier [117]	IFT [48]	HGD [92]	AutoHAS
LR	MNet2 (S0)	44.6±0.6	12.3±8.7	6.1±4.5	N/A	29.6±2.1	44.8±0.4
	MNet2 (T0)	52.4±0.5	17.5±3.0	14.3±20.0	N/A	33.0±4.4	52.0±0.2
	MNet2	66.8±0.2	38.9±5.6	49.0±3.6	N/A	49.1±4.6	66.9±0.1
	ENet-A0	60.8±0.0	46.6±1.2	50.8±0.8	N/A	50.0±1.4	61.0±0.0
	ResNet-18	67.6±0.1	60.4±1.2	63.5±0.2	N/A	56.3±0.5	67.9±0.2
	ResNet-50	74.8±0.1	67.2±0.1	71.1±0.3	N/A	62.3±0.3	75.2±0.1
L2	MNet2 (S0)	44.6±0.6	45.9±0.7	46.3±0.2	46.2±0.1	N/A	46.3±0.1
	MNet2 (T0)	52.4±0.5	52.2±0.0	52.4±0.4	52.5±0.2	N/A	53.5±0.3
	MNet2	66.8±0.2	66.4±0.8	67.0±0.2	66.4±0.2	N/A	67.5±0.1
	ENet-A0	60.8±0.0	60.0±2.0	62.0±0.2	61.1±0.2	N/A	62.2±0.1
	ResNet-18	67.6±0.1	67.9±0.2	67.6±0.1	66.6±0.3	N/A	67.9±0.0
	ResNet-50	74.8±0.1	75.0±0.1	74.8±0.1	73.1±0.4	N/A	75.0±0.1
LR +L2	MNet2 (S0)	44.6±0.6	13.1±10.9	15.2±7.3	N/A	N/A	45.7±0.3
	MNet2 (T0)	52.4±0.5	29.3±20.6	30.2±15.9	N/A	N/A	53.8±0.2
	MNet2	66.8±0.2	21.6±15.1	25.2±14.6	N/A	N/A	67.3±0.1
	ENet-A0	60.8±0.0	47.3±4.7	49.3±2.4	N/A	N/A	61.5±0.1
	ResNet-18	67.6±0.1	54.2±8.5	53.5±0.5	N/A	N/A	67.8±0.0
	ResNet-50	74.8±0.1	67.4±4.7	66.7±1.9	N/A	N/A	74.8±0.1
A+LR +L2	MNet2 (S0)	44.6±0.6	22.4±12.4	25.4±4.1	46.4±0.4	N/A	47.5±0.3
	ENet-A0	60.8±0.0	53.4±5.7	56.4±3.9	61.8±0.5	N/A	62.9±0.2

Table 4.3 : We compare four AutoML algorithms on four search spaces.

hyperparameters is similar or slightly lower than the default. It might because the default choices are close to the optimal solution in the current HAS search space on CIFAR-10.

Model	Parameters	FLOPs	Train Time	Searching Methods			
	(MB)	(M)	(seconds)	RS / Vizier	IFT-Neumann	HGD	AutoHAS
MNet2 (S0)	1.49	35.0	2.0e3	1.9e4 (9.4×)	2.0e3 (1.0×)	2.6e3 (1.3×)	2.8e3 (1.4×)
MNet2 (T0)	1.77	89.5	2.1e3	2.0e4 (9.3×)	2.5e3 (1.2×)	4.1e3 (2.0×)	2.4e3 (1.2×)
MNet2	3.51	307.3	2.4e3	1.8e4 (7.5×)	5.7e3 (2.3×)	2.5e3 (1.1×)	4.7e3 (1.9×)
ENet-A0	2.17	76.2	1.4e3	1.2e4 (8.7×)	2.2e3 (1.6×)	1.9e3 (1.4×)	2.2e3 (1.6×)
ResNet-18	11.69	1818	2.0e3	1.9e4 (9.6×)	2.7e3 (1.4×)	2.2e3 (1.1×)	2.2e3 (1.1×)
ResNet-50	25.56	4104	2.6e3	2.0e4 (7.6×)	2.9e3 (1.1×)	2.8e3 (1.1×)	2.8e3 (1.1×)

Table 4.4 : We report the computational costs of each model and the searching costs of each AutoML algorithm on ImageNet.

	Birdsnap	CIFAR-10	CIFAR-100	Cars
default	51.7±0.7	93.9±0.2	75.7±0.2	72.0±0.8
GDAS (Arch) [11]	55.8±0.6	93.5±0.0	76.4±0.5	77.0±2.5
AutoHAS (HP)	54.4±0.7	93.9±0.1	76.0±0.1	77.7±0.1
AutoHAS (HP+Arch)	56.5±0.9	93.7±0.3	76.0±0.4	80.3±0.5

Table 4.5 : We use *AutoHAS* to search for hyperparameters (HP), architectures (Arch), and both hyperparameters and architectures (HP+Arch).

4.3.4 AutoHAS for SQuAD

To further validate the generalizability of *AutoHAS*, we also conduct experiments on a reading comprehension dataset in the NLP domain, i.e., SQuAD 1.1 [182]. We pretrain a $\text{BERT}_{\text{LARGE}}$ model following [178] and then apply *AutoHAS* when fine-tuning it on SQuAD 1.1. In particular, we search the per-step learning rate and weight decay of Adam. For *AutoHAS*, we split the training set of SQUAD 1.1 into 80% for training and 20% validation. In Figure 4.4, we show the results on the dev set, and compare the default setup in [178] with hyperparameters found

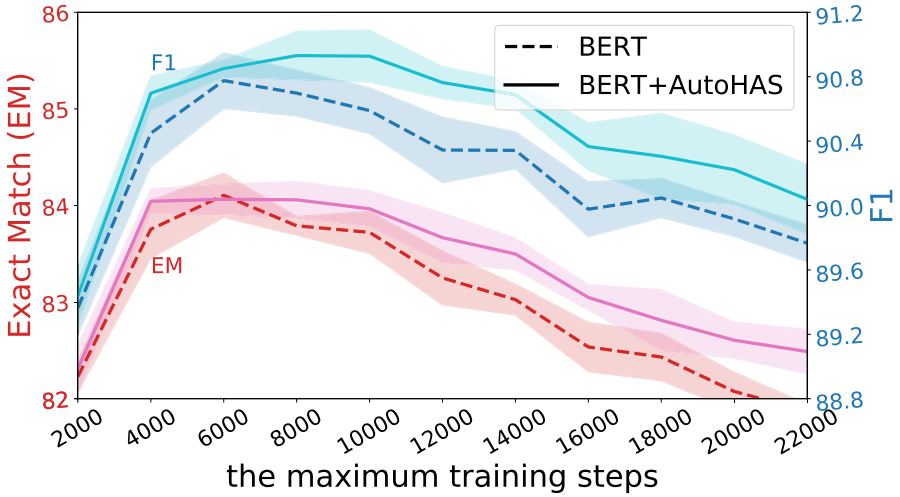


Figure 4.4 : Performance comparison on SQuAD 1.1.

by *AutoHAS*. We vary the fine-tuning steps from 2K to 22K and each setting is run 5 times. We can see that *AutoHAS* is superior to the default hyperparameters under most of the circumstances, in terms of both F1 and exact match (EM) scores. Notably, the average gain on F1 over all the steps is 0.3, which is highly nontrivial.

4.4 Conclusion

In this chapter, we study the joint search of hyperparameters and architectures. Our framework overcomes the unrealistic assumptions in NAS that the relative ranking of models' performance is primarily affected by their architecture. To address the challenge of joint search, we proposed *AutoHAS*, i.e., an efficient and differentiable searching algorithm for both hyperparameters and architecture. *AutoHAS* represents the hyperparameters and architectures in a unified way to handle the mixture of categorical and continuous values of the search space. *AutoHAS* shares weights across all hyperparameters and architectures, which enable it to search efficiently over the joint large search space. Experiments on both large-scale vision and NLP datasets demonstrate the effectiveness of *AutoHAS*. With the help of this

general framework AutoHAS, we achieved the fifth objective (Obj-5, “Generalize the proposed efficient algorithm from the search for the architecture only to also include the hyperparameters.”).

Chapter 5

Benchmarks for Automated Deep Learning

5.1 Introduction

In this chapter, we study the foundation of AutoDL, aiming to answer the third question described in Chapter 1 – (Q-3 "How could we enable reproducible AutoDL?"). By building a large-scale architecture dataset and benchmarking tens of AutoDL algorithms, we will achieve the objective Obj-6 identified in Chapter 1.

Recently, a variety of NAS algorithms have been increasingly proposed. While these NAS techniques are methodically designed and show promising improvements, many setups in their algorithms are different. (1) Different search space is utilized, e.g., range of macro skeletons of the whole architecture [54, 60] and a different operation set for the micro cell within the skeleton [29], etc. (2) After a good architecture is selected, various strategies can be employed to train this architecture and report the performance, e.g., different data augmentation [184, 185], different regularization [54], different scheduler [40], and different selections of hyperparameters [73, 9]. (3) The validation set for testing the performance of the selected architecture is not split in the same way [28, 29]. These discrepancies cause a problem when comparing the performance of various NAS algorithms, making it difficult to conclude their relative contributions.

In response to this challenge, NAS-Bench-101 [98] and NAS-HPO-Bench [95] were proposed. However, some NAS algorithms cannot be applied *directly* on NAS-Bench-101, and NAS-HPO-Bench only has 144 candidate architectures which may be insufficient to comprehensively evaluate NAS algorithms. NAS-Bench-1shot1 [99]

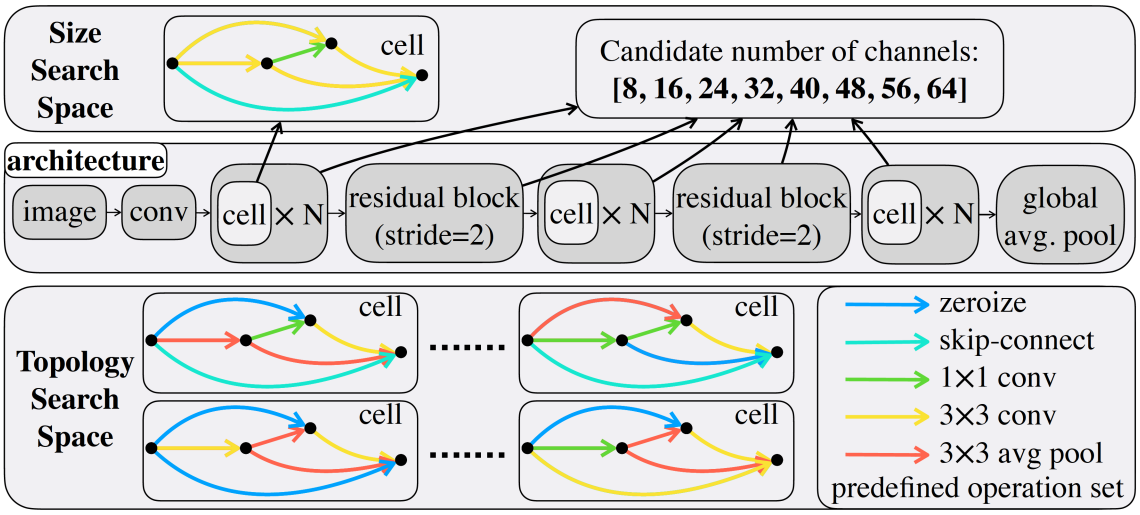


Figure 5.1 : The overview of the topology and size search space in *NATS-Bench*.

reuses the NAS-Bench-101 dataset with some modification to analyse the one-shot NAS methods. The aforementioned works have mainly focused on the architecture topology. However, the architecture size*, which significantly affects a model's performance, is not considered in the existing benchmarks.

To enlarge the scope of these benchmarks and towards better reproducibility of NAS methods, we propose *NATS-Bench* with (1) a topology search space $\mathcal{S}_t$ to be applicable for all NAS methods and (2) a size search space $\mathcal{S}_s$ that supplements the lack of analysis for the architecture size. As shown in the middle part of Figure 5.1, each architecture consists of a predefined skeleton with a stack of the searched cells. Each cell is represented as a densely-connected directed acyclic graph (DAG) as shown in the bottom section of Figure 5.1. The node represents the sum of the feature maps and each edge is associated with an operation transforming the feature maps from the source node to the target node.

*Some papers may use size to indicate the number of parameters of a neural network. In this manuscript, the terminology "architecture size" or "size" refer to the number of channels in each layer following [8].

In $\mathcal{S}_t$, we search for the operation assigned on each edge, and thus its size is related to the number of nodes defined for the DAG and the size of the operation set. We choose 4 nodes and 5 representative operation candidates for the operation set, which generates a total search space of 15,625 cells/architectures. In $\mathcal{S}_s$, we search for the number of channels in each layer (i.e., convolution, cell, or block). We pre-define 8 candidates for the number of channels, which generates a total search space of $8^5 = 32768$ architectures (see more details in the top part of Figure 5.1. Each architecture in $\mathcal{S}_t$ and $\mathcal{S}_s$ is trained multiple times on three different datasets. The training log and performance of each architecture are provided for each run. The training accuracy/test accuracy/training loss/test loss after every training epoch for each architecture plus the number of parameters and floating point operations (FLOPs) are accessible.

NATS-Bench has shown its value in the field of NAS research. (1) It provides the *first* benchmark to study the architecture size. (2) It provides a unified benchmark for most up-to-date NAS algorithms including all cell-based NAS methods. With *NATS-Bench*, researchers can focus on designing robust searching algorithm while avoiding tedious hyperparameter tuning of the searched architecture. Thus, *NATS-Bench* provides a relatively fair benchmark for the comparison of different NAS algorithms. (3) It provides the full training log of each architecture. Unnecessary repetitive training procedure of each selected architecture can be avoided [73, 33] so that researchers can target on the essence of NAS, i.e., search algorithm. Another benefit is that the validation time for NAS largely decreases when testing in *NATS-Bench*, which provides a computational power friendly environment for more participation in NAS. (4) It provides results of each architecture on multiple datasets. The model transferability can be thoroughly evaluated for most NAS algorithms. (5) In *NATS-Bench*, we provide systematic analysis of the proposed search space. We also evaluate 13 recent advanced NAS algorithms including rein-

forcement learning (RL)-based methods, evolutionary strategy (ES)-based methods, differentiable-based methods, etc. Our empirical analysis can bring some insights to the future designs of NAS algorithms.

	#Unique DNNs	#Datasets	Diagnostic Information	Search Space	Supported NAS algorithms RL	ES	Diff.	HPO
NAS-Bench-101	423k	1	✗	topology	partial	partial	none	most
$\mathcal{S}_t$ in *NATS-Bench*	6.5k	3	fine-grained accuracy	topology	all	all	all	most
$\mathcal{S}_s$ in *NATS-Bench*	32.8k	3	and loss, parameters, etc	size	all	all	most	most

Table 5.1 : We summarize the important characteristics of NAS-Bench-101 and *NATS-Bench*.

5.2 *NATS-Bench*

Our *NATS-Bench* is algorithm-agnostic. Put simply, it is applicable to almost any up-to-date NAS algorithm. In this section, we will briefly introduce our *NATS-Bench*. The search space of *NATS-Bench* is inspired by cell-based NAS algorithms (Section 5.2.1). *NATS-Bench* evaluates each architecture on three different datasets (Section 5.2.2). All implementation details of *NATS-Bench* are introduced in Section 5.2.3. *NATS-Bench* also provides some diagnostic information which can be used for potentially better designs of future NAS algorithms (discussed in Section 5.2.4).

5.2.1 Architectures in the Search Space

Macro Skeleton. Our search space follows the design of its counterpart as used in the recent neural cell-based NAS algorithms [28, 54, 29]. As shown in the middle part of Figure 5.1, the skeleton is initiated with one 3-by-3 convolution with 16 output channels and a batch normalization layer [171]. The main body of

the skeleton includes three stacks of cells, connected by a residual block. All cells in an architecture has the same topology. The intermediate residual block is the basic residual block with a stride of 2 [20], which serves to down-sample the spatial size and double the channels of an input feature map. The shortcut path in this residual block consists of a 2-by-2 average pooling layer with stride of 2 and a 1-by-1 convolution. The skeleton ends up with a global average pooling layer to flatten the feature map into a feature vector. The classification uses a fully connected layer with a softmax layer to transform the feature vector into the final prediction.

The Topology Search Space $\mathcal{S}_t$. The topology search space is inspired by the popular cell-based NAS algorithms [11, 28, 54]. Since all cells in an architecture have the same topology, an architecture candidate in $\mathcal{S}_t$ corresponds to a different cell, which is represented as a densely connected DAG. The densely connected DAG is obtained by assigning a direction from the i-th node to the j-th node ($i < j$) for each edge in an undirected complete graph. Each edge in this DAG is associated with an operation transforming the feature map from the source node to the target node. All possible operations are selected from a predefined operation set, as shown in Figure 5.1(bottom-right). In our *NATS-Bench*, the predefined operation set $\mathcal{O}$ has $L = 5$ representative operations: (1) zeroize, (2) skip connection, (3) 1-by-1 convolution, (4) 3-by-3 convolution, and (5) 3-by-3 average pooling layer. The convolution in this operation set is an abbreviation of an operation sequence of ReLU, convolution, and batch normalization. The DAG has $V = 4$ nodes, where each node represents the sum of all feature maps transformed through the associated operations of the edges pointing to this node. We choose $V = 4$ to allow the search space to contain basic residual block-like cells, which require 4 nodes. Densely connected DAG does not restrict the searched topology of the cell to be densely connected, since we include zeroize in the operation set, which is an operation of dropping the associated edge. We do not impose the constraint on the maximum number of edges [98], and thus $\mathcal{S}_t$

	optimizer	Nesterov	learning rate (LR)	momentum	weight decay	batch size	norm	random flip	random crop	epoch
value	SGD	✓	cosine decay LR from 0.1 to 0	0.9	0.0005	256	✓	p=0.5	✓	12

Table 5.2 : The training hyperparameters $\mathcal{H}^0$ for all candidate architectures in the size search space $\mathcal{S}_s$ and the topology search space $\mathcal{S}_t$.

is applicable to most NAS algorithms, including all cell-based NAS algorithms. For each architecture in $\mathcal{S}_t$, each cell is stacked $N = 5$ times, with the number of output channels set to 16, 32 and 64 for the first, second and third stages, respectively.

The Size Search Space $\mathcal{S}_s$. The size search space is inspired by transformable architecture search methods [8, 186, 151]. In the size search space, every stack in each architecture is constructed by stacking $N = 1$ cell. All cells in every architecture have the same topology, which is the best one in $\mathcal{S}_t$ on the CIFAR-100 dataset. Each architecture candidate in $\mathcal{S}_s$ has a different configuration regarding the number of channels in each layer.[†] We build the size search space $\mathcal{S}_s$ to include the largest number of channels in $\mathcal{S}_t$. Therefore, the number of channels in each layer is chosen from {8, 16, 24, 32, 40, 48, 56, 64}. Therefore, the size search space $\mathcal{S}_s$ has $8^5 = 32768$ architecture candidates.

5.2.2 Datasets

We train and evaluate each architecture on CIFAR-10, CIFAR-100 [132], and ImageNet-16-120 [187]. We choose these three datasets because CIFAR and ImageNet [133] are the most popular image classification datasets.

We split each dataset into training, validation and test sets to provide a consistent training and evaluation settings for previous NAS algorithms [28]. Most NAS methods use the validation set to evaluate architectures after the architecture is op-

[†]A layer could be the stem 3-by-3 convolutional layer, the cell, or the residual block.

timized on the training set. The validation performance of the architectures serves as the supervision signals to update the searching algorithm. The test set is to evaluate the performance of each searching algorithm by comparing the indicators (e.g., accuracy, #parameters, speed) of their selected architectures. Previous methods use different splitting strategies, which may result in various searching costs and unfair comparisons. We hope to use the proposed splits to unify the training, validation and test sets for a fairer comparison.

CIFAR-10: It is a standard image classification dataset and consists of 60K 32×32 colour images in 10 classes. The original training set contains 50K images, with 5K images per class. The original test set contains 10K images, with 1K images per class. Due to the need of validation set, we split all 50K training images in CIFAR-10 into two groups. Each group contains 25K images with 10 classes. We regard the first group as the new training set and the second group as the validation set.

CIFAR-100: This dataset is just like CIFAR-10. It has the same images as CIFAR-10 but categorizes each image into 100 fine-grained classes. The original training set on CIFAR-100 has 50K images, and the original test set has 10K images. We randomly split the original test set into two groups of equal size — 5K images per group. One group is regarded as the validation set, and another one is regarded as the new test set.

ImageNet-16-120: We build ImageNet-16-120 from the down-sampled variant of ImageNet (ImageNet16×16). As indicated in [187], down-sampling images in ImageNet can largely reduce the computation costs for optimal hyperparameters of some classical models while maintaining similar searching results. [187] down-sampled the original ImageNet to 16×16 pixels to form ImageNet16×16, from which we select all images with label $\in [1, 120]$ to construct ImageNet-16-120. In sum, ImageNet-16-120 contains 151.7K training images, 3K validation images, and 3K test images

with 120 classes.

By default, in this chapter, "the training set", "the validation set", "the test set" indicate the new training, validation, and test sets, respectively.

5.2.3 Architecture Performance

Training Architectures. In order to unify the performance of every architecture, we provide the performance of every architecture in our search space. In our *NATS-Bench*, we follow previous literature to set up the hyperparameters and training strategies [54, 40, 20]. We train each architecture with the same strategy, which is shown in Table 5.2. For simplification, we denote all hyperparameters for training a model as a set $\mathcal{H}$. We use $\mathcal{H}^0$, $\mathcal{H}^1$, and $\mathcal{H}^2$ to denote the three kinds of hyperparameters that we use. Specifically, we train each architecture via Nesterov momentum SGD, using the cross-entropy loss. We set the weight decay to 0.0005 and decay the learning rate from 0.1 to 0 with a cosine annealing [40]. We use the same $\mathcal{H}^0$ on different datasets, except for the data augmentation which is slightly different due to the image resolution. On the CIFAR datasets, we use the random flip with probability of 0.5, the random crop 32×32 patch with 4 pixels padding on each border, and the normalization over RGB channels. On ImageNet-16-120, we use a similar strategy but with random crop 16×16 patch and 2 pixels padding on each border. In $\mathcal{H}^0$, we train each architecture by 12 epochs, which can be used in bandit-based algorithms [77, 188]. Since 12 epochs are not sufficient to evaluate the relative ranking of different architectures, we train each candidate with more epochs ($\mathcal{H}^1$ and $\mathcal{H}^2$) to obtain a more accurate ranking. $\mathcal{H}^1$ and $\mathcal{H}^2$ are the same as $\mathcal{H}^0$ but use 200 epochs and 90 epochs, respectively. In *NATS-Bench*, we apply $\mathcal{H}^0$ and $\mathcal{H}^1$ on the topology search space $\mathcal{S}_t$; and we apply $\mathcal{H}^0$ and $\mathcal{H}^2$ on the size search space $\mathcal{S}_s$.

Metrics. We train each architecture with different random seeds on different

datasets. We evaluate each architecture α after every training epoch. *NATS-Bench* provides the training, validation, and test loss as well as accuracy. Users can easily use our API to query the results of each trial of α, which has negligible computational costs. In this way, researchers could significantly speed up their searching algorithm on these datasets and focus solely on the essence of NAS.

5.2.4 Diagnostic Information

Validation accuracy is a commonly used supervision signal for NAS. However, considering the expensive computational costs for evaluating the architecture, the signal is too sparse. In our *NATS-Bench*, we also provide some additional diagnostic information in a form of extra statistics obtained during training of each architecture. Collecting these statistics almost involves no extra computation cost but may provide insights for better designs and training strategies of different NAS algorithms, such as platform-aware NAS [60], accuracy prediction [83], mutation-based NAS [82, 59], etc.

Architecture Computational Costs: *NATS-Bench* provides three computation metrics for each architecture — the number of parameters, FLOPs, and latency. Algorithms that focus on searching architectures with computational constraints, such as models on edge devices, can use these metrics directly in their algorithm designs without extra calculations. We also provide the training time and evaluation time for each architecture.

Fine-grained training and evaluation information. *NATS-Bench* tracks the changes in loss and accuracy of every architecture after every training epoch. These fine-grained training and evaluation information often shows the trends related to the architecture performance and could help with identifying some attributes of the model, such as the speed of convergence, the stability, the over-fitting or under-fitting levels, etc. These attributes may benefit the designs of NAS algorithms.

Besides, some methods learn to predict the final accuracy of an architecture based on the results of few early training epochs [83]. These algorithms can be trained faster and the performance of the accuracy prediction can be evaluated using the fine-grained evaluation information.

Parameters of the optimized architecture. Our *NATS-Bench* releases the trained parameters for each architecture. This can provide ground truth label for hypernetwork-based NAS methods [130, 55], which learn to generate parameters of an architecture. Other methods mutate an architecture to become another one [76, 82]. With *NATS-Bench*, researchers could directly use the off-the-shelf parameters instead of training them from scratch and analyze how to transfer parameters from one architecture to another.

5.2.5 What/Who can Benefit from *NATS-Bench*?

Our *NATS-Bench* provides a unified NAS library for the community and can benefit NAS algorithms from the perspective of both performance and efficiency. NAS has been dominated by multi-fidelity based methods [79, 77, 83, 76], which learn to search based on an approximation of the performance of each candidate in order to accelerate searching. Running algorithms on our *NATS-Bench* can reduce the approximation to an accurate performance via only querying from the database. This can avoid sub-optimal training because of the inaccurate estimation of the performance as well as accelerate the training into seconds. Meanwhile, with the provision of our diagnostic information, such as latency, algorithms trained with such extra pieces of information can directly fetch them from our codebase with negligible efforts. Meanwhile, the designs of NAS algorithms can also have more diversity with the benefit of the diagnostic information and more potential designs will be discussed in Section 5.5.

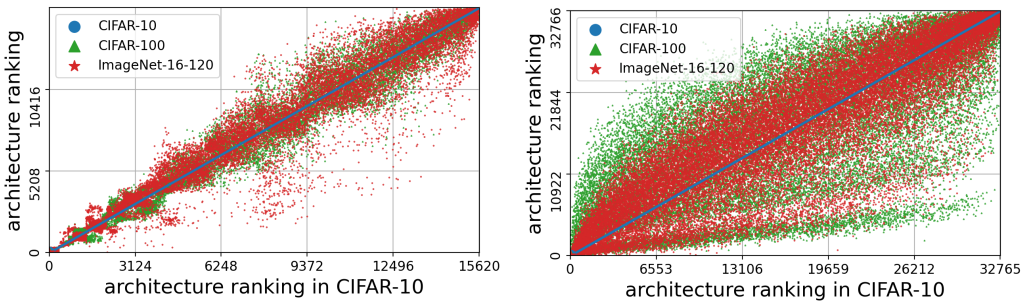
In the NAS community, there has been a growing attention to the field of joint

searching for both topology and size [174, 189]. By benchmarking their sub-modules for either topology or size on *NATS-Bench*, it may help researchers to understand the effectiveness of the sub-modules and give inspirations for ongoing and future research which lies in this intersection.

NATS-Bench provides a unified codebase – a NAS library – to make the benchmarking as fair as possible. In this codebase, we share the code implementation for different algorithms as much as possible. For example, the super network for weight-sharing methods is reused; the data pipelines for different methods are reused; the interface of training, forwarding, optimizing for different algorithms is kept the same. We demonstrate, using 13 state-of-the-art NAS algorithms applied on *NATS-Bench*, how the process has been unified through an easy-to-use API. The implementation difference between DARTS [28] and GDAS [11] is only less than 20 lines of code. Our library reduces the effect caused by the implementation difference when comparing different methods. It is also easy to implement new NAS algorithms by reusing and extending our library. More detailed engineering designs can be found in the documentation of our released codes. As this part is beyond the scope of this manuscript, we do not introduce it here.

5.3 Analysis of *NATS-Bench*

An Overview of Architecture Performance: The performance of each architecture in both search spaces $\mathcal{S}_t$ and $\mathcal{S}_s$ is shown in Figure 5.3. The training and test accuracy with respect to the number of parameters and number of FLOPs are shown in each column, respectively. Results show that a different number of parameters or FLOPs will affect the performance of the architectures, which indicates that the choices of operations are essential in NAS. We also observe that the performance of the architecture can vary even when the number of parameters or FLOPs stays the same.

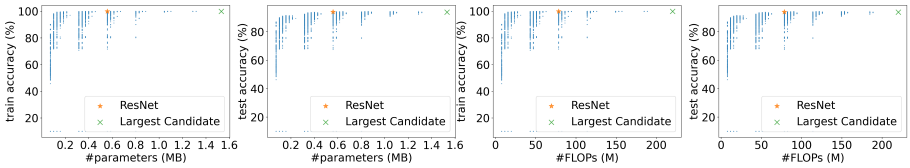


(a) The relative ranking for the topology search space $\mathcal{S}_t$.

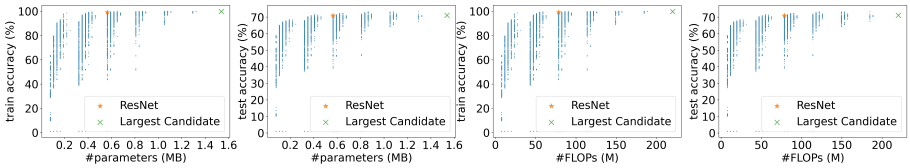
(b) The relative ranking for the size search space $\mathcal{S}_s$.

Figure 5.2 : The ranking of each architecture on three datasets, sorted by the ranking in CIFAR-10.

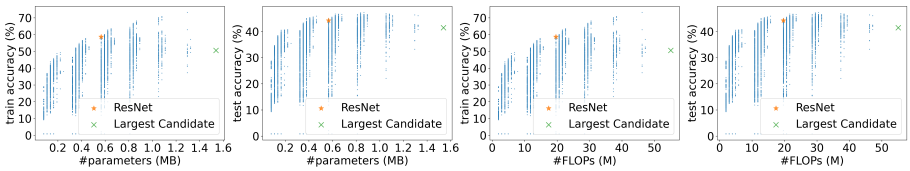
These observations indicate the importance of how the operations are connected and how the number of channels is set. We compare all architectures in $\mathcal{S}_t$ and $\mathcal{S}_s$ with some classical human-designed architectures (orange star marks in Figure 5.3). (I) Compared to candidates in $\mathcal{S}_t$, ResNet shows competitive performance in three datasets, however, it still has room to improve, i.e., about 2% compared to the best architecture in CIFAR-100 and ImageNet-16-120, about 1% compared to the best one with the same amount of parameters in CIFAR-100 and ImageNet-16-120. (II) In many vision tasks, pyramid structure has shown a surprising robustness and accuracy [190, 191]. Regarding the parameters vs. the accuracy, the candidates in $\mathcal{S}_s$ with a pyramid structure are far from the Pareto optimality. Regarding the FLOPs vs. the accuracy, the candidates in $\mathcal{S}_s$ with a pyramid structure are close to the Pareto optimality.



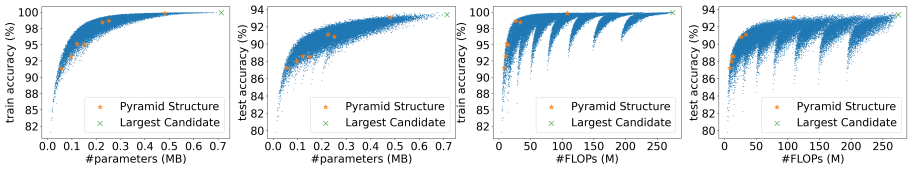
(a) Results of all candidates in the topology search space $\mathcal{S}_t$ on CIFAR-10.



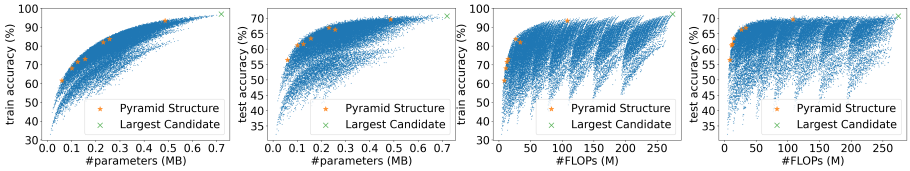
(b) Results of all candidates in the topology search space $\mathcal{S}_t$ on CIFAR-100.



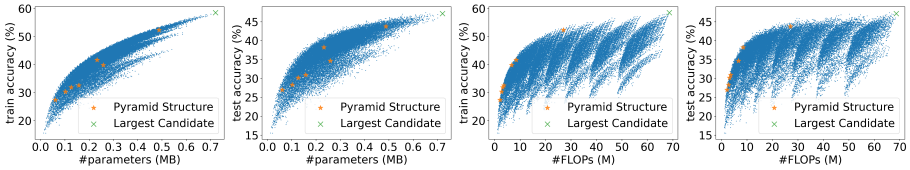
(c) Results of all candidates in the topology search space $\mathcal{S}_t$ on ImageNet-16-120.



(d) Results of all candidates in the size search space $\mathcal{S}_s$ on CIFAR-10.



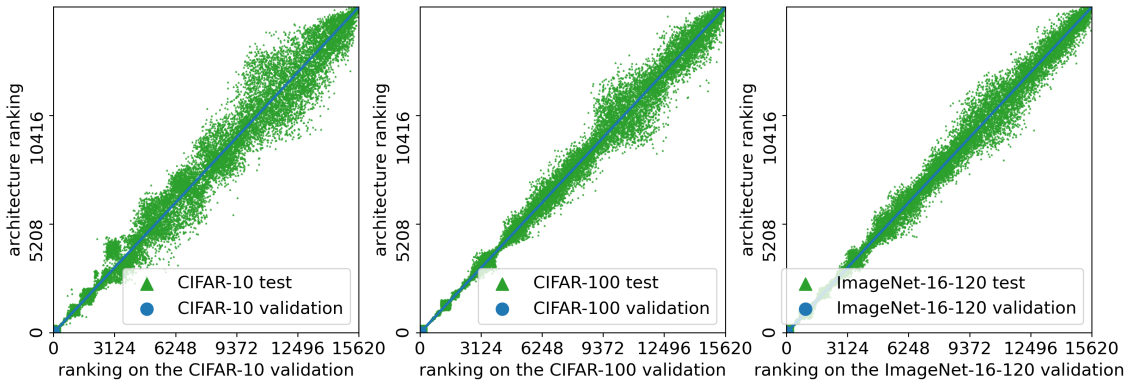
(e) Results of all candidates in the size search space $\mathcal{S}_s$ on CIFAR-100.



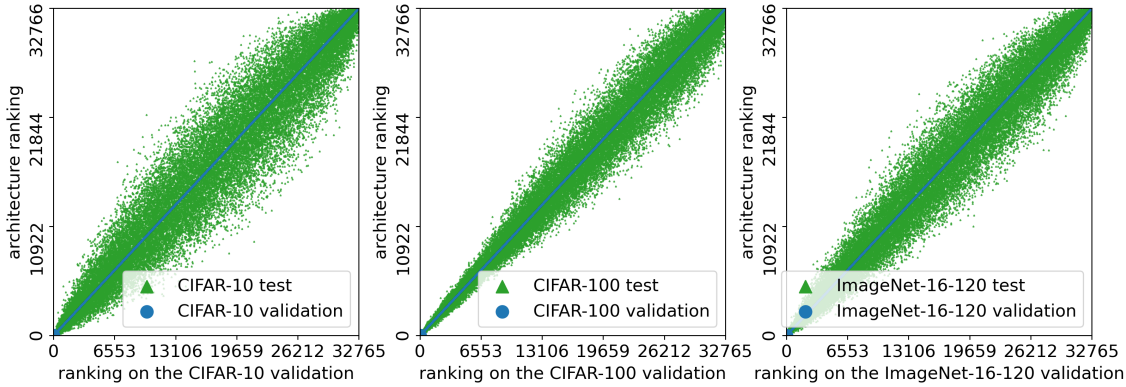
(f) Results of all candidates in the size search space $\mathcal{S}_s$ on ImageNet-16-120.

Figure 5.3 : The training and test accuracy vs. #parameters and FLOPs.

Architecture Ranking on Three Datasets: The ranking of every architecture in our search space is shown in Figure 5.2, where the architectures ranked



(a) The relative ranking for the topology search space $\mathcal{S}_t$.



(b) The relative ranking for the size search space $\mathcal{S}_s$.

Figure 5.4 : The correlation between the validation accuracy and the test accuracy for all architecture candidates in $\mathcal{S}_t$ and $\mathcal{S}_s$.

in CIFAR-10 (x-axis) are shown in relation to their respective ranks in CIFAR-100 and ImageNet-16-120 (y-axis), indicated by green and red markers respectively. The performance of the architectures in $\mathcal{S}_t$ shows a generally consistent ranking over the three datasets with slightly different variance, which serves to test the generality of the searching algorithm. In contrast, the ranking of architecture candidates in $\mathcal{S}_s$ is quite different. It indicates that the optimal architecture sizes on three datasets are different.

We compute the validation as well as the test accuracy after training with $\mathcal{H}^1$ and $\mathcal{H}^2$ on $\mathcal{S}_t$ and $\mathcal{S}_s$, respectively. Figure 5.4 visualizes their correlation. It shows the

relative ranking obtained from the validation accuracy is similar to that obtained using the test accuracy. Thus, it guarantees the upper bounds of the NAS algorithms, because the brute-force strategy can find an architecture that can almost achieve the highest test accuracy.

We also show the correlation coefficient across different datasets in Figure 5.5. The correlation dramatically decreases as we only pick the top performing architecture candidates. When we directly transfer the best architecture in one dataset to another (i.e. a vanilla strategy), it can not 100% secure a good performance. This phenomena is a call for better transferable NAS algorithms instead of using the vanilla strategy.

5.4 Benchmark

5.4.1 Bi-level Optimization of NAS

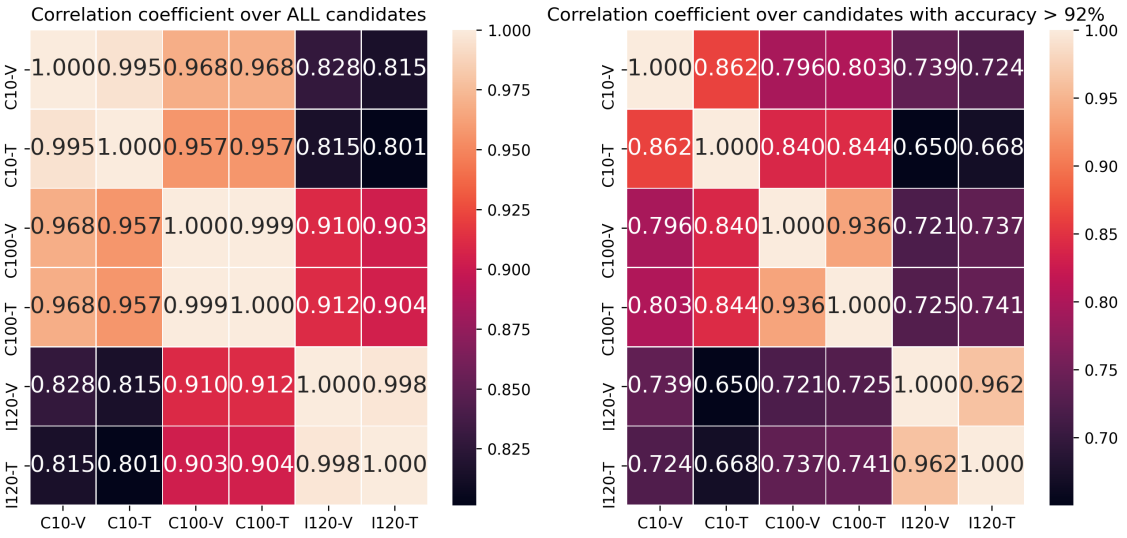
NAS aims to find architecture α among the search space $\mathcal{S}$ so that this found α achieves a high performance on the validation set. This problem can be formulated as a bi-level optimization problem:

$$\begin{aligned} &\arg\min_{\alpha \in \mathcal{S}} \mathcal{L}(\alpha, \omega_\alpha^*, \mathcal{D}_{val}) \\ &\text{s.t. } \omega_\alpha^* = \arg\min\nolimits_\omega \mathcal{L}(\alpha, \omega, \mathcal{D}_{train}), \end{aligned} \tag{5.1}$$

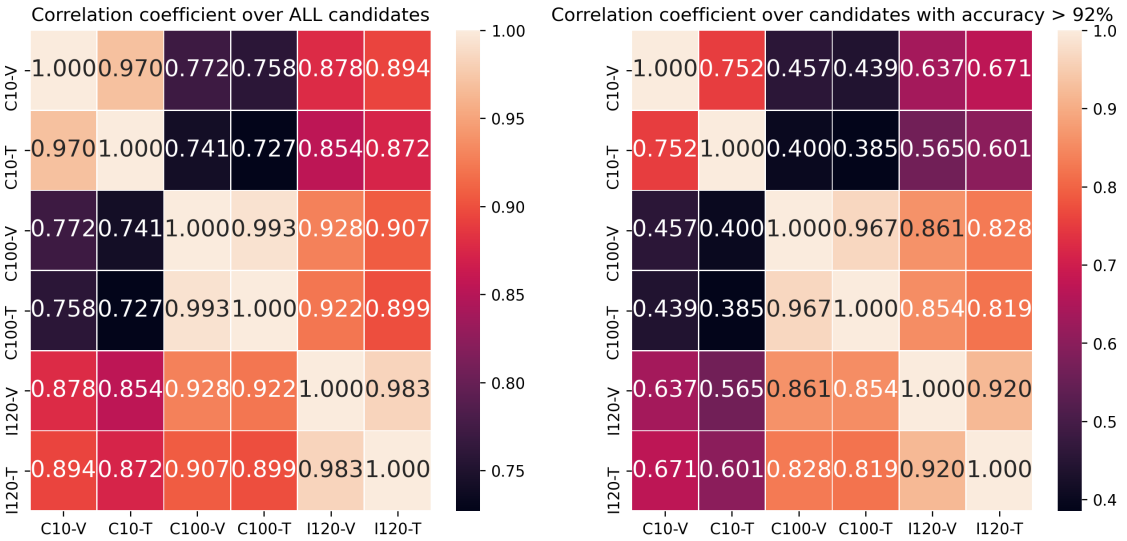
where $\mathcal{L}$ indicates the objective function (e.g., cross-entropy loss). $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$ denote the training data and the validation data, respectively. In the typical NAS setting, after an architecture α is found, α will be evaluated on the test data $\mathcal{D}_{test}$ to figure out its real performance.

5.4.2 Experimental Setup

We evaluate **13** recent, state-of-the-art searching methods on our *NATS-Bench*, which can serve as baselines for future NAS algorithms in our dataset. Specifically,



(a) The correlation coefficient for the topology search space $\mathcal{S}_t$.



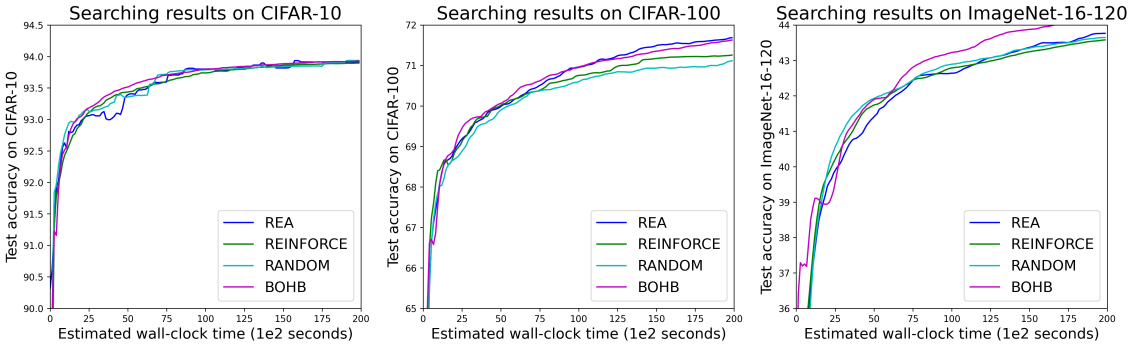
(b) The correlation coefficient for the size search space $\mathcal{S}_s$.

Figure 5.5 : The correlation coefficient between accuracy on different datasets.

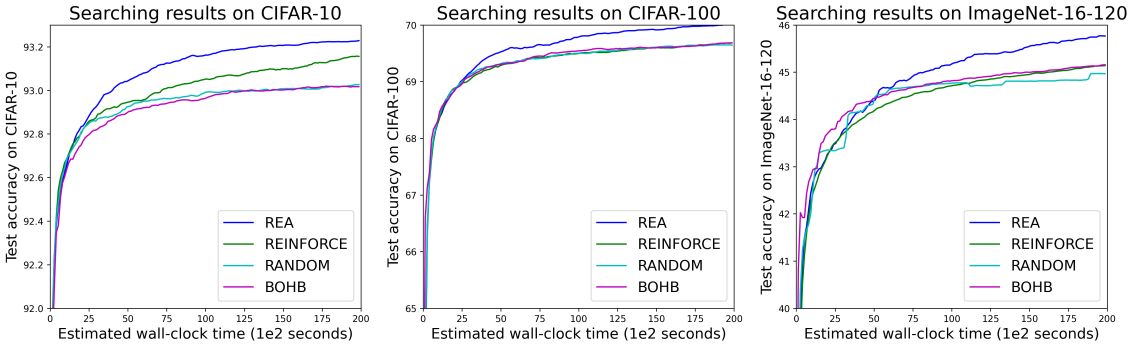
we evaluate some typical NAS algorithms: (I) Random Search algorithms, e.g., random search (RANDOM) [24], random search with parameter sharing (RSPS) [79]. (II) ES methods, e.g., REA [76]. (III) RL algorithms, e.g., REINFORCE [192], ENAS [29]. (IV) Differentiable algorithms. e.g., first order DARTS (DARTS-V1) [28], second order DARTS (DARTS-V2), GDAS [11], SETN [9], TAS [8], FBNet-

Accelerate the search procedure	RANDOM, REINFORCE, REA, and BOHB
Accelerate the evaluation procedure	all NAS methods

Table 5.3 : The utility of our *NATS-Bench* for different NAS algorithms.



(a) Results of NAS algorithms without weight sharing in the topology search space $\mathcal{S}_t$.



(b) Results of NAS algorithms without weight sharing in the size search space $\mathcal{S}_s$.

Figure 5.6 : The test accuracy of the searched architecture over time.

V2 [189], TuNAS [193]. (V) HPO methods, e.g., BOHB [77].

Among them, RANDOM, REA, REINFORCE, and BOHB are multi-trial based methods. They can be used to search on both $\mathcal{S}_t$ and $\mathcal{S}_s$ search spaces. Especially, using our API, we can accelerate them to be executed in seconds as shown in Table 5.3.

Other methods are weight-sharing based methods, in which the evaluation pro-

cedure can be accelerated by using our API. Notably, DARTS, GDAS, SETN are specifically designed for the topology search space $\mathcal{S}_t$. TAS, FBNet-V2, and TuNAS can be used on the size search space $\mathcal{S}_s$.

5.4.3 Experimental Results

Multi-trial based Methods

We follow the suggested hyperparameters in their original papers to run each method on our topology search space $\mathcal{S}_t$ and size search space $\mathcal{S}_s$. We run each experiment 500 times on three datasets and setup a maximum time budget as 2e4 seconds. Every 100 seconds, each method can let us know the current searched architecture candidate. We use the hyperparameters $\mathcal{H}^0$ (12 epochs) to obtain a validation accuracy for each trial. This validation accuracy serves as the supervision/feedback signal for these multi-trial based methods. For BOHB, given its current budget for a trial, it can early stop before fully training the model using 12 epochs. We plot the averaged accuracy of this searched architecture candidate over 500 runs in Figure 5.6. Each sub-figure corresponds to one dataset and a search space. For example, in the middle of Figure 5.6b, we search on CIFAR-100 and show the test accuracy of the searched architecture on CIFAR-100.

Observations on the topology search space $\mathcal{S}_t$. (1) On CIFAR-10, most methods have similar performance. (2) On CIFAR-100, REA is similar to BOHB, which outperforms REINFORCE; and RANDOM is the worst among them. (3) On ImageNet-16-120, BOHB significantly outperforms the other methods. It may be caused by the dynamic budget mechanism for each trial in BOHB, which allows to traverse more architecture candidates.

Observations on the size search space $\mathcal{S}_s$. (1) REA significantly outperforms the other methods on all datasets in the size search space $\mathcal{S}_s$. (2) On CIFAR-100 and ImageNet-16-120, results of BOHB, REINFORCE, and RANDOM are similar.

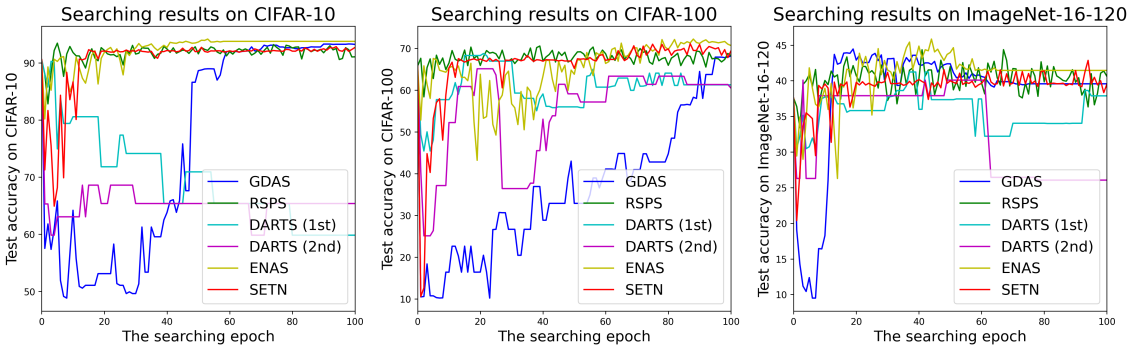
(3) On CIFAR-10, REINFORCE is better than BOHB and RANDOM. (4) As the searching time goes, the searched architecture by REA gradually matches the best one, while the other methods need much more time to catch up with REA. (5) Figure 5.3 implies a simple prior for $\mathcal{S}_s$: without the constraint of model cost, the larger model tends to have higher accuracy. By visualising the searched architecture, REA can quickly fit this prior while the other methods do not.

Given the flexibility and robustness of REA, we would recommend choosing REA as a searching algorithm if the computational resources are sufficient.

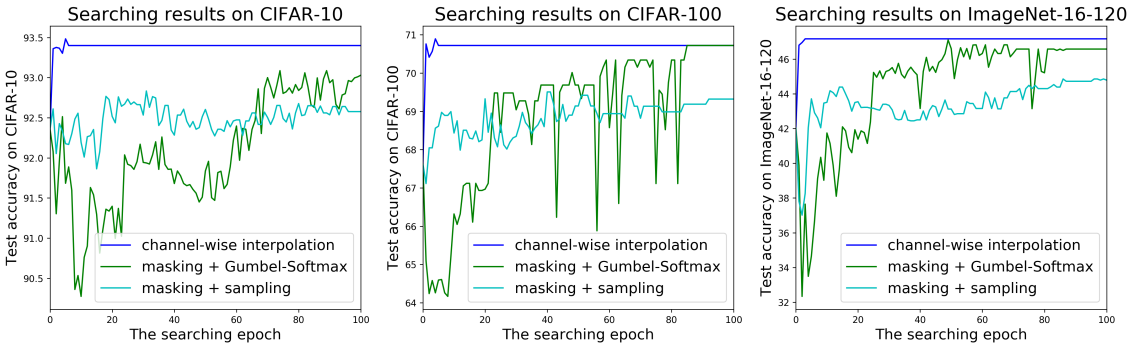
Weight-sharing based Methods

To compare weight-sharing based methods as fairly as possible, we keep the same hyperparameters concerned with the optimising of the shared weights for different methods. For other hyperparameters, e.g., hyperparameters for optimising the controller in ENAS or hyperparameters for optimising the architectural parameters in DARTS/GDAS, we use the same values as introduced in their original papers. In this way, we can focus on evaluating the core and unique modules in each searching algorithm. We setup the total number of epochs to 100 for search, and compare results of their searched architecture candidates after each search epoch. We run each experiment three times and report the average results in Figure 5.7a for the topology search space and in Figure 5.7b for the size search space.

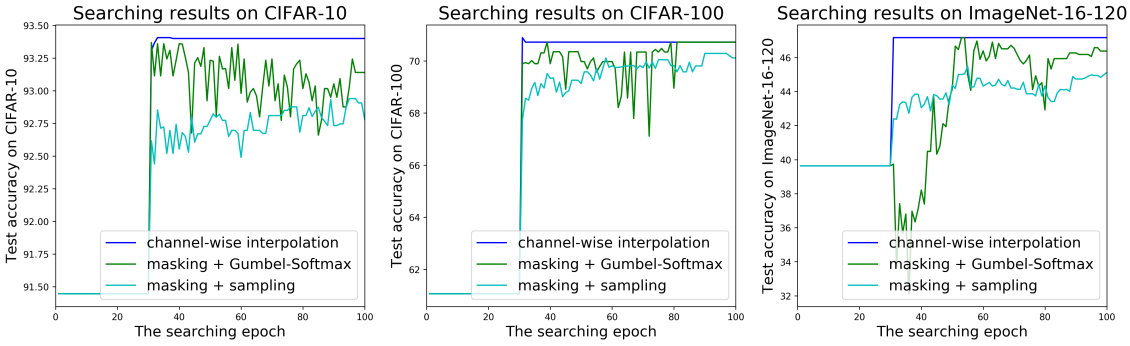
Observations on the topology search space $\mathcal{S}_t$. (1) On CIFAR-10, DARTS-V1 and DARTS-V2 quickly converge to find the architecture having many skip connections, which performs poorly. However, on CIFAR-100 and ImageNet-16-120, they perform relatively well. This is because the significantly increased searching data on CIFAR-100 and ImageNet-16-120 over CIFAR-10 alleviate the problem of incorrect gradient estimation in bi-level optimization. (2) RSPS, ENAS, and SETN converge quickly and are robust on three datasets. During their searching proce-



(a) Results of weight-sharing based methods in the topology search space $\mathcal{S}_t$.



(b) Results of weight-sharing based methods in the size search space $\mathcal{S}_s$. We do not add any #FLOPs or #parameters constraint for these searching methods.



(c) We use the same setting as that in Figure 5.7b, while we apply the warm-up strategy that is similar to [189, 193].

Figure 5.7 : The test accuracy of the searched architecture after each epoch.

dure, they will randomly sample some architecture candidates, evaluate them using the shared weights, and select the candidate with the highest validation accuracy. Such strategy is more robust than using the arg max over the learned architecture

parameters in [28, 11]. (3) The searched architecture of GDAS slowly converges to the similar one as ENAS and SETN.

Some observations on $\mathcal{S}_t$ are different from those in our preliminary version. It is because some hyperparameters changed following either suggestions from the authors or better strategies found in our experiments. Especially, we would like to highlight some useful strategies for weight-sharing based methods: (1) always use batch statistics for the batch normalization layer. (2) do not learn the scale and shift parameters of the batch normalization layer. (3) during the evaluation procedure of RSPS, ENAS, and SETN, the average accuracy for a large batch of validation data is sufficient to approximate the average accuracy on the whole validation set. In our experiments, we use the batch size of 512 for evaluation.

Observations on the size search space $\mathcal{S}_s$. We abstract the three kinds of strategies to search for #channels:

- Using channel-wise interpolation to explicitly compare two different #channels [8].
- Using the masking mechanism to represent different candidate #channels and optimize its distribution via Gumbel-Softmax [189].
- Using the masking mechanism to represent different candidate #channels and optimize its distribution via REINFORCE [193].

We indicate these strategies as "channel-wise interpolation", "masking + Gumbel-Softmax", and "masking + sampling" in Figure 5.7b and Figure 5.7c. "channel-wise interpolation" can quickly find much better model than masking-based strategies. It might because that the interpolation strategy allows us to implicitly evaluate and compare two candidate #channels in each layer during each search step. In contrast, the masking strategies can only evaluate one candidate during each search

step. As shown in Figure 5.7c, we also try the same filter warm-up strategy as [193]. Overall speaking, the performance of the discovered architecture is improved by the warm-up strategy.

Since the original hyperparameters of [8, 189, 193] are designed for ImageNet and joint searching of filters and operations, they might be sub-optimal for the settings in *NATS-Bench*. In addition, we re-implement these algorithms based on our codebase. They may have some differences compared to the original implementation due to the different search spaces, libraries, etc. Therefore, it is under investigation of whether our empirical observations can generalize to other scenarios or not.

Weight-sharing vs. Multi-trial based Methods

The weight-sharing based methods and multi-trial based methods have their unique advantages and disadvantages. Multi-trial based methods can theoretically find the best architecture as long as the proxy task is accurate, and the number of trials is large enough. However, their prohibitive computational cost has motivated researchers to design efficient weight-sharing based algorithms. However, sharing weights sacrifices the accuracy of each architecture candidate. As the search space increases, the shared weights are usually not able to distinguish the performance of different candidates.

Clarification. We have tried our best to implement each method using their reported best experimental set ups. However, please be aware that some algorithms might still result in sub-optimal performance since their hyperparameters might not be optimal for our *NATS-Bench*. We empirically found that some NAS algorithms are sensitive to some hyperparameters, and we have tried to compare them in as fair a way as possible. If researchers can provide better results with different hyperparameters, we are happy to update the benchmarks according to the new experimental results. We also welcome more NAS algorithms to be tested on our

dataset and would be happy to include them accordingly.

5.5 Discussion

How to avoid over-fitting on *NATS-Bench*? Our *NATS-Bench* provides a benchmark for NAS algorithms, aiming to provide a fair and computationally cost-friendly environment to the NAS community. The trained architecture and the easy-to-access performance of each architecture might provide some insidious ways for designing algorithms to over-fit the best architecture in our *NATS-Bench*. Thus, we propose some rules to follow in order to achieve the original intention of *NATS-Bench*, a fair and efficient benchmark.

1. No regularization for a specific operation. Since the best architecture is known in our benchmark, specific designs to fit the structural attributes of the best performing architecture constitute one of the insidious ways to fit our *NATS-Bench*. For example, as mentioned in Section 5.4, we found that the best architecture with the same number of parameters for CIFAR10 on *NATS-Bench* is ResNet. Restrictions on the number of residual connections is a way to over-fit the CIFAR10 benchmark. While this can give a good result on this benchmark, the searching algorithm might not generalize to other benchmarks.

2. Use the same meta hyperparameter for different datasets and search spaces in NATS-Bench. The searching algorithm has some meta hyperparameter that controls the behaviour of search. For example, the temperature τ in GDAS or the band width factor in BOHB. Using the same meta hyperparameter could evaluate the robustness of the searching algorithm and prevent it from over-fitting to a specific dataset.

3. Use the provided performance. The training strategy affects the performance of the architecture. We suggest to stick to the performance provided in our bench-

mark even if it is feasible to use other $\mathcal{H}$ to get a better performance. This provides a fair comparison with other algorithms.

4. Report results of multiple searching runs. Since our benchmark can help to largely decrease the computational cost for a number of algorithms, multiple searching runs, which give stable results of the searching algorithm with acceptable time cost, are strongly recommended.

5.6 Conclusion

In this chapter, we proposed an algorithm-agnostic NAS benchmark (NATS-Bench) with information of 15,625 neural cell candidates for architecture topology and 32,768 for architecture size on three datasets. We also provided 13 NAS baselines in a single codebase, and comprehensively analyzed these popular NAS algorithms. The proposed NATS-Bench facilitated the scientific research in AutoDL and thus solved the sixth objective in this thesis (Obj-6, "Build large-scale architecture datasets to encourage reproducible neural architecture search in the AutoDL community.").

Chapter 6

Conclusions and Future Work

Please recall the three research questions proposed at the beginning of this thesis: (1) How could we reduce the massive computational cost of AutoDL algorithms? (2) How could we design a unified and efficient AutoDL framework? (3) How could we enable reproducible AutoDL? Throughout the thesis from Chapter 3 to Chapter 5, a preliminary answer to them has been provided.

To be specific, in Chapter 3, we showed that the parameter sharing approach significantly reduces the massive training cost for a neural architecture search, and a learnable distribution over the search space can help reduce the number of trials that are required for AutoDL. Furthermore, a gradient-based architecture sampler is proposed to improve this line of research further – achieving the fastest AutoDL algorithm for architecture search in the year 2018 [11, 8]. In Chapter 4, we generalized the efficient AutoDL framework proposed in Chapter 3 to handle booth architecture and hyperparameters. The new AutoDL framework is efficient and also general to searching for various different components in a deep learning pipeline [5]. In Chapter 5, we built a large-scale architecture dataset to facilitate the AutoDL research [1]. Along with this dataset, we built a unified AutoDL codebase, including over ten popular NAS algorithms. With the help of this dataset and codebase, 13 recent NAS algorithms have been systemically evaluated. Some interesting observations revealed how to choose AutoDL algorithms for different requirements.

In summary, the identified objectives in Chapter 1 have been completed as follows:

To Obj-1 In Chapter 2, we briefly review the history of AutoDL. For sub-directions in AutoDL, the NAS and HPO algorithms are analysed in Section 2.1 and Section 4.2.1, respectively. The AutoDL benchmarks and software are discussed in Section 2.3 and Section 2.4, respectively. Lastly, we discussed the motivation of our proposed algorithms and the pros/cons of our algorithms vs. other alternatives in Section 2.5.

To Obj-2 In Section 3.2, we have proposed an efficient NAS algorithm, named GDAS, that can discover a robust CNN on CIFAR within four GPU hours.

To Obj-3 In Section 3.2, we additionally applied the proposed GDAS to a search space RNN for NLP tasks. The improved performance demonstrates the generalization ability of GDAS.

To Obj-4 The initial version of GDAS can only be applied to searching for the architecture topology. However, apart from topology, the architecture size is crucial to trade-off the efficiency and accuracy of an architecture. In Section 3.3, we equipped the GDAS with channel-wise interpolation to handle the architecture size, and it achieved the SoTA accuracy on the network pruning task.

To Obj-5 To further extend the scope of search from architectures to hyperparameters, we upgraded our GDAS algorithm to AutoHAS in Chapter 4. AutoHAS is a general framework that can efficiently and jointly search for both architectures and hyperparameters.

To Obj-6 Due to the importance of architecture topology and size, we have built an algorithm-agnostic NAS benchmark to facilitate the research of architecture search. Our benchmark provides the information of 15,625 neural cell candidates for architecture topology and 32,768 for architecture size on three vision datasets. It has shown great value to the AutoDL community [1].

In this thesis, we mainly focus on the efficiency and reproducibility of AutoDL algorithms. Even with them, there is still a long way to go to apply AutoDL in the production environment. Our proposed algorithm relies on the predefined search space. When moving to imperfect search space, the search performance would dramatically degrade. Another limitation comes from the engineering side. Since more different aspects are incorporated into the search space, we need to handle both conditional space, categorical space, continuous space, etc. It is not easy to implement AutoDL algorithms that can flexibly switch between different kinds of spaces. Therefore, to further improve AutoDL – pushing its boundary, we need to consider the following aspects.

- AutoDL from Scratch. Existing AutoDL algorithms heavily rely on the expert knowledge to manually design their search space. To start search on a new application, it takes considerable effort to research the handcrafted design and design the search space to align with this application. Could we automatically design such prior knowledge for AutoDL in future?

- AutoDL Systems. AutoDL is a promising paradigm for automating these choices. Current deep learning software libraries, however, are quite limited in handling the dynamic interactions among the components of AutoDL. For example, efficient NAS algorithms typically require an implementation coupling between the search space and search algorithm, the two key components in AutoDL. Furthermore, implementing a complex search flow, such as searching architectures within a loop of searching hardware configurations, is difficult. To summarize, changing the search space, search algorithm, or search flow in current deep learning libraries usually requires a significant change in the program logic. This requirement urges us to develop new systems to accommodate AutoDL.

Bibliography

[1] X. Dong, L. Liu, K. Musial, and B. Gabrys, "NATS-Bench: Benchmarking nas algorithms for architecture topology and size," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2021, doi:10.1109/TPAMI.2021.3054824.

[2] X. Dong, Y. Yang, S.-E. Wei, X. Weng, Y. Sheikh, and S.-I. Yu, "Supervision by registration and triangulation for landmark detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020, doi:10.1109/TPAMI.2020.2983935.

[3] X. Dong, L. Zheng, F. Ma, Y. Yang, and D. Meng, "Few-example object detection with model communication," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 41, no. 7, pp. 1641–1654, July 2019, doi:10.1109/TPAMI.2018.2844853.

[4] X. Dong, Y. Yan, M. Tan, Y. Yang, and I. W. Tsang, "Late fusion via subspace search with consistency preservation," *IEEE Transactions on Image Processing (TIP)*, vol. 28, no. 1, pp. 518–528, Jan 2019.

[5] X. Dong, M. Tan, A. W. Yu, D. Peng, B. Gabrys, and Q. V. Le, "AutoHAS: Efficient hyperparameter and architecture search," in *International Conference on Learning Representations Workshop (ICLR-W)*, 2021.

[6] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020.

[7] D. Peng, X. Dong, E. Real, M. Tan, Y. Lu, G. Bender, H. Liu, A. Kraft, C. Liang, and Q. Le, "PyGlove: Symbolic programming for automated machine learning," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[8] X. Dong and Y. Yang, "Network pruning via transformable architecture search," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2019, pp. 760–771.

[9] ——, "One-shot neural architecture search via self-evaluated template network," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 3681–3690.

[10] ——, "Teacher supervises students how to learn from partially labeled images for facial landmark detection," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2019.

[11] ——, "Searching for a robust neural architecture in four gpu hours," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 1761–1770.

[12] X. Dong, S.-I. Yu, X. Weng, S.-E. Wei, Y. Yang, and Y. Sheikh, "Supervision-by-Registration: An unsupervised approach to improve the precision of facial landmark detectors," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 360–368.

[13] X. Dong, Y. Yan, W. Ouyang, and Y. Yang, "Style aggregated network for facial landmark detection," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[14] X. Dong, J. Huang, Y. Yang, and S. Yan, "More is less: A more complicated network with less inference complexity," in *Proc. of the IEEE Conference on*

Computer Vision and Pattern Recognition (CVPR), 2017, pp. 5840–5848.

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105.

[17] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[18] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 39, no. 6, pp. 1137–1149, 2017.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[21] S. Arlot, A. Celisse *et al.*, "A survey of cross-validation procedures for model selection," *Statistics Surveys*, vol. 4, pp. 40–79, 2010.

[22] T. Elsken, J. H. Metzen, F. Hutter *et al.*, "Neural architecture search: A survey." *J. Mach. Learn. Res.*, vol. 20, no. 55, pp. 1–21, 2019.

[23] M. Feurer and F. Hutter, "Hyperparameter optimization," in *Automated Machine Learning*. Springer, Cham, 2019, pp. 3–33.

[24] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research (JMLR)*, vol. 13, no. Feb, pp. 281–305, 2012.

[25] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[26] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[27] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[28] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *International Conference on Learning Representations (ICLR)*, 2019.

[29] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *The International Conference on Machine Learning (ICML)*, 2018, pp. 4095–4104.

[30] R. D. King, C. Feng, and A. Sutherland, "Statlog: comparison of classification algorithms on large real-world problems," *Applied Artificial Intelligence an International Journal*, vol. 9, no. 3, pp. 289–333, 1995.

[31] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated Machine Learning*. Springer, 2019.

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[33] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2017.

[34] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *The Journal of Machine Learning Research (JMLR)*, vol. 21, no. 243, pp. 1–18, 2020.

[35] D. Michie, D. J. Spiegelhalter, C. Taylor *et al.*, "Machine learning," *Neural and Statistical Classification*, vol. 13, no. 1994, pp. 1–298, 1994.

[36] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 40, no. 4, pp. 834–848, 2017.

[37] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei, "Auto-DeepLab: Hierarchical neural architecture search for semantic image segmentation," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 82–92.

[38] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[39] G. Shani and A. Gunawardana, "Evaluating recommendation systems," in *Recommender Systems Handbook*. Springer, 2011, pp. 257–297.

[40] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," in *International Conference on Learning Representations (ICLR)*, 2017.

[41] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2011, pp. 2546–2554.

[42] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2012, pp. 2951–2959.

[43] D. Maclaurin, D. Duvenaud, and R. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *The International Conference on Machine Learning (ICML)*, 2015, pp. 2113–2122.

[44] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyper-parameter optimization of deep neural networks by extrapolation of learning curves," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 3460–3468.

[45] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2017.

[46] Z. Liu, Z. Xu, S. Rajaa, M. Madadi, J. C. J. Junior, S. Escalera, A. Pavao, S. Treguer, W.-W. Tu, and I. Guyon, "Towards automated deep learning: Analysis of the autodl challenge series 2019," in *NeurIPS 2019 Competition and Demonstration Track*. PMLR, 2020, pp. 242–252.

[47] E. Real, C. Liang, D. R. So, and Q. V. Le, "Automl-zero: Evolving machine learning algorithms from scratch," in *The International Conference on Machine Learning (ICML)*, 2020.

[48] J. Lorraine, P. Vicol, and D. Duvenaud, "Optimizing millions of hyperparameters by implicit differentiation," in *International Conference on Artificial*

Intelligence and Statistics (AISTATS), 2020.

[49] H. Pham and Q. V. Le, "Autodropout: Learning dropout patterns to regularize deep networks," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[50] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *The International Conference on Machine Learning (ICML)*, 2017, pp. 2430–2439.

[51] J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber, "Evolving memory cell structures for sequence learning," in *International Conference on Artificial Neural Networks*, 2009, pp. 755–764.

[52] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *The International Conference on Machine Learning (ICML)*, 2015, pp. 2342–2350.

[53] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *The International Conference on Machine Learning (ICML)*, 2017, pp. 2902–2911.

[54] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 8697–8710.

[55] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "SMASH: one-shot model architecture search through hypernetworks," in *International Conference on Learning Representations (ICLR)*, 2018.

[56] D. Ha, A. Dai, and Q. V. Le, "HyperNetworks," in *International Conference on Learning Representations (ICLR)*, 2017.

[57] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[58] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," in *The International Conference on Machine Learning (ICML)*. PMLR, 2018, pp. 678–687.

[59] T. Chen, I. Goodfellow, and J. Shlens, "Net2net: Accelerating learning via knowledge transfer," in *International Conference on Learning Representations (ICLR)*, 2016.

[60] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2820–2828.

[61] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *The International Conference on Machine Learning (ICML)*, 2018, pp. 550–559.

[62] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Uncertainty in Artificial Intelligence*, 2020, pp. 367–377.

[63] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 1284–1293.

[64] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "NAS-FPN: Learning scalable feature pyramid architecture for object detection," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 7036–7045.

[65] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," in *International Conference on Learning Representations (ICLR)*, 2017.

[66] X. Du, T.-Y. Lin, P. Jin, G. Ghiasi, M. Tan, Y. Cui, Q. V. Le, and X. Song, "SpineNet: Learning scale-permuted backbone for recognition and localization," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 11 592–11 601.

[67] B. Ru, P. Esperanca, and F. Carlucci, "Neural architecture generator optimization," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[68] H. Liu, A. Brock, K. Simonyan, and Q. V. Le, "Evolving normalization-activation layers," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[69] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," in *International Conference on Learning Representations (ICLR)*, 2018.

[70] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[71] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[72] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations (ICLR)*, 2019.

[73] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[74] D. So, Q. Le, and C. Liang, "The evolved transformer," in *The International Conference on Machine Learning (ICML)*, 2019, pp. 5877–5886.

[75] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 10 428–10 436.

[76] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2019, pp. 4780–4789.

[77] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *The International Conference on Machine Learning (ICML)*, 2018, pp. 1436–1445.

[78] C. Lyle, L. Schut, R. Ru, Y. Gal, and M. van der Wilk, "A bayesian perspective on training speed and model selection," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[79] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *The Conference on Uncertainty in Artificial Intelligence (UAI)*, 2019.

[80] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2020.

[81] S. Xie, H. Zheng, C. Liu, and L. Lin, "SNAS: stochastic neural architecture

search," in *International Conference on Learning Representations (ICLR)*, 2019.

[82] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2018, pp. 2787–2794.

[83] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," in *International Conference on Learning Representations Workshop (ICLR-W)*, 2018.

[84] F. Hutter, "Automated configuration of algorithms for solving hard computational problems," Ph.D. dissertation, University of British Columbia, 2009.

[85] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International Conference on Learning and Intelligent Optimization*, 2011, pp. 507–523.

[86] R. Kohavi and G. H. John, "Automatic parameter selection by minimizing estimated error," in *Machine Learning Proceedings*, 1995, pp. 304–312.

[87] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[88] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable bayesian optimization using deep neural networks," in *The International Conference on Machine Learning (ICML)*, 2015, pp. 2171–2180.

[89] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algo-

rithms," in *The SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013, pp. 847–855.

[90] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan *et al.*, "Population based training of neural networks," *arXiv preprint arXiv:1711.09846*, 2017.

[91] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research (JMLR)*, vol. 18, no. 1, pp. 6765–6816, 2017.

[92] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online learning rate adaptation with hypergradient descent," in *International Conference on Learning Representations (ICLR)*, 2018.

[93] F. Pedregosa, "Hyperparameter optimization with approximate gradient," in *The International Conference on Machine Learning (ICML)*, 2016, pp. 737–746.

[94] A. Shaban, C.-A. Cheng, N. Hatch, and B. Boots, "Truncated back-propagation for bilevel optimization," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2019, pp. 1723–1732.

[95] A. Klein and F. Hutter, "Tabular benchmarks for joint architecture and hyperparameter optimization," *arXiv preprint arXiv:1905.04970*, 2019.

[96] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search," in *The International Conference on Machine Learning (ICML) Workshop*, 2018.

[97] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and efficient object detection," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 10 781–10 790.

[98] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *The International Conference on Machine Learning (ICML)*, 2019, pp. 7105–7114.

[99] A. Zela, J. Siems, and F. Hutter, "Nas-bench-1shot1: Benchmarking and dissecting one shot neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020.

[100] K. Yu, C. Sciuto, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020.

[101] ——, "Evaluating the search phase of neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020.

[102] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, and E. Burnaev, "NAS-Bench-NLP: neural architecture search benchmark for natural language processing," *arXiv preprint arXiv:2006.07116*, 2020.

[103] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," *arXiv preprint arXiv:2008.09777*, 2020.

[104] C. Li, Z. Yu, Y. Fu, Y. Zhang, Y. Zhao, H. You, Q. Yu, Y. Wang, and Y. Lin, "HW-NAS-Bench: Hardware-aware neural architecture search benchmark," in *International Conference on Learning Representations (ICLR)*, 2021.

[105] A. Mehrotra, A. G. Ramos, S. Bhattacharya, Ł. Dudziak, R. Vipperla, T. Chau, M. S. Abdelfattah, S. Ishtiaq, and N. D. Lane, "NAS-Bench-ASR:

Reproducible neural architecture search for speech recognition," in *International Conference on Learning Representations (ICLR)*, 2021.

[106] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a cpu and gpu math expression compiler," in *Proceedings of the Python for scientific computing conference (SciPy)*, 2010, pp. 18–24.

[107] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *The {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[108] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2019, pp. 8024–8035.

[109] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *The ACM Multimedia Conference*, 2014, pp. 675–678.

[110] S. Tokui, "Chainer: A powerful, flexible and intuitive framework of neural networks," 2018.

[111] R. Frostig, M. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," in *Conference on Machine Learning and Systems (MLSys)*, 2018.

[112] J. Shen, P. Nguyen, Y. Wu, Z. Chen, M. X. Chen, Y. Jia, A. Kannan, T. Sainath, Y. Cao, C.-C. Chiu *et al.*, "Lingvo: a modular and scalable frame-

work for sequence-to-sequence modeling," *arXiv preprint arXiv:1902.08295*, 2019.

[113] "Gin-config," https://github.com/google/gin-config.

[114] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2015, pp. 2962–2970.

[115] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka," *The Journal of Machine Learning Research (JMLR)*, vol. 18, no. 1, pp. 826–830, 2017.

[116] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: efficient and robust automated machine learning," in *Automated Machine Learning*. Springer, 2019, pp. 113–134.

[117] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *The SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[118] "Neural network intelligence," https://github.com/microsoft/nni.

[119] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[120] R. Negrinho, D. Patil, N. Le, D. Ferreira, M. Gormley, and G. Gordon, "Towards modular and programmable architecture search," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2019, pp. 13 715–13 725.

[121] "Keras tuner," https://github.com/keras-team/keras-tuner.

[122] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *The SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019, pp. 1946–1956.

[123] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *The SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[124] W. Wen, H. Liu, Y. Chen, H. Li, G. Bender, and P.-J. Kindermans, "Neural predictor for neural architecture search," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2020, pp. 660–676.

[125] Y. Shu, W. Wang, and S. Cai, "Understanding architectures learnt by cell-based neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020.

[126] E. J. Gumbel, "Statistical theory of extreme values and some practical applications," *NBS Applied Mathematics Series (AMS)*, vol. 33, 1954.

[127] C. J. Maddison, D. Tarlow, and T. Minka, "A* sampling," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2014, pp. 3086–3094.

[128] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," in *International Conference on Learning Representations (ICLR)*, 2017.

[129] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[130] C. Zhang, M. Ren, and R. Urtasun, "Graph hypernetworks for neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2019.

[131] R. Luo, F. Tian, T. Qin, and T.-Y. Liu, "Neural architecture optimization," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2018, pp. 7827–7838.

[132] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[133] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[134] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[135] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *International Conference on Learning Representations (ICLR)*, 2017.

[136] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[137] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[138] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6848–6856.

[139] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[140] J. G. Zilly, R. K. Srivastava, J. Koutnik, and J. Schmidhuber, "Recurrent highway networks," in *The International Conference on Machine Learning (ICML)*, 2017, pp. 4189–4198.

[141] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing LSTM language models," in *International Conference on Learning Representations (ICLR)*, 2018.

[142] Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen, "Breaking the softmax bottleneck: A high-rank rnn language model," in *International Conference on Learning Representations (ICLR)*, 2018.

[143] H. Inan, K. Khosravi, and R. Socher, "Tying word vectors and word classifiers: A loss framework for language modeling," in *International Conference on Learning Representations (ICLR)*, 2017.

[144] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 1990, pp. 598–605.

[145] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations (ICLR)*, 2015.

[146] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2015, pp. 1135–1143.

[147] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep

neural networks," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1389–1397.

[148] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, pp. 2234–2240.

[149] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *International Conference on Learning Representations (ICLR)*, 2017.

[150] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2736–2744.

[151] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and H. Song, "AMC: Automl for model compression and acceleration on mobile devices," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2018, pp. 183–202.

[152] J. M. Alvarez and M. Salzmann, "Learning the number of neurons in deep networks," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2016, pp. 2270–2278.

[153] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *The Conference on Neural Information Processing Systems Workshop (NeurIPS-W)*, 2014.

[154] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," in *International Conference on Learning Representations (ICLR)*, 2017.

[155] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of

network pruning," in *International Conference on Learning Representations (ICLR)*, 2019.

[156] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 38, no. 10, pp. 1943–1955, 2016.

[157] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 1993, pp. 164–171.

[158] M. Figurnov, M. D. Collins, Y. Zhu, L. Zhang, J. Huang, D. Vetrov, and R. Salakhutdinov, "Spatially adaptive computation time for residual networks," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1039–1048.

[159] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research (JMLR)*, vol. 18, no. 1, pp. 6869–6898, 2017.

[160] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, and Y. Gal, "An empirical study of binary neural networks' optimisation," in *International Conference on Learning Representations (ICLR)*, 2019.

[161] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *The ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[162] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization," in *International Conference on Learning Representations (ICLR)*, 2018.

[163] E. Tartaglione, S. Lepsøy, A. Fiandrotti, and G. Francini, "Learning sparse neural networks via sensitivity-driven regularization," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2018, pp. 3878–3888.

[164] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *The Conference on Neural Information Processing Systems (NeurIPS)*, 2016, pp. 2074–2082.

[165] J. Ye, X. Lu, Z. Lin, and J. Z. Wang, "Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers," in *International Conference on Learning Representations (ICLR)*, 2018.

[166] Y. He, P. Liu, Z. Wang, and Y. Yang, "Pruning filter via geometric median for deep convolutional neural networks acceleration," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4340–4349.

[167] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi, "MorphNet: Fast & simple resource-constrained structure learning of deep networks," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 1586–1595.

[168] B. Minnehan and S. Andreas, "Cascaded projection: End-to-end network compression and acceleration," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 10 715–10 724.

[169] M. Jaderberg, K. Simonyan, A. Zisserman *et al.*, "Spatial transformer networks," in *The Conference on Neural Information Processing Systems*

(NeurIPS), 2015, pp. 2017–2025.

[170] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 37, no. 9, pp. 1904–1916, 2015.

[171] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[172] J. Yu and T. Huang, "Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers," *arXiv preprint arXiv:1903.11728*, 2019.

[173] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[174] H. Cai, C. Gan, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," in *International Conference on Learning Representations (ICLR)*, 2020.

[175] L. Xie and A. Yuille, "Genetic CNN," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1379–1388.

[176] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 10 734–10 742.

[177] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *The International Conference on Machine Learning (ICML)*, 2019, pp. 6105–6114.

[178] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *The Conference of the Association for Computational Linguistics (ACL)*, 2019, pp. 4171–4186.

[179] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 248–255.

[180] T. Berg, J. Liu, S. Woo Lee, M. L. Alexander, D. W. Jacobs, and P. N. Belhumeur, "Birdsnap: Large-scale fine-grained visual categorization of birds," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014, pp. 2011–2018.

[181] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization," in *International IEEE Workshop on 3D Representation and Recognition (3dRR)*, 2013.

[182] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016, pp. 2383–2392.

[183] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[184] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "Dropblock: A regularization method for convolutional networks," in *The Conference on Neural Information Processing*

Systems (NeurIPS), 2018, pp. 10 727–10 737.

[185] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," in *International Conference on Learning Representations (ICLR)*, 2018.

[186] J. Yu and T. S. Huang, "Universally slimmable networks and improved training techniques," in *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 1803–1811.

[187] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "A downsampled variant of imagenet as an alternative to the cifar datasets," *arXiv preprint arXiv:1707.08819*, 2017.

[188] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research (JMLR)*, vol. 18, no. 1, pp. 6765–6816, 2018.

[189] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen *et al.*, "FBNetV2: Differentiable neural architecture search for spatial and channel dimensions," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 12 965–12 974.

[190] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2117–2125.

[191] D. Han, J. Kim, and J. Kim, "Deep pyramidal residual networks," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5927–5935.

[192] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[193] G. Bender, H. Liu, B. Chen, G. Chu, S. Cheng, P.-J. Kindermans, and Q. V. Le, "Can weight sharing outperform random architecture search? an investigation with tunas," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 14 323–14 332.